(2)

Carnegie-Mellon University
Software Engineering Institute

Unit Testing and Analysis

Curriculum Module SEI-CM-9-1.2

Unit Testing and Analysis

SEI Curriculum Module SEI-CM-9-1.2

April 1989

Larry J. Morell
College of William and Mary

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.
Approved for public release. Distribution unlimited.

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Unit Testing and Analysis

Acknowledgements

I want to thank the members of the Education Program at the Software Engineering Institute for their support and encouragement during the writing of this module. Gary Ford and Lionel Deimel always asked the right questions at the right time. In his handling of administrative details, Albert Johnson enabled me to concentrate on the task at hand. Director of Education Norm Gibbs provided many opportunities for interaction with other members of the Institute.

Two student assistants were instrumental in constructing the bibliography and are much deserving of thanks. Suzanne Woolf performed the library searches, and Dana Hausman spent many hours typing abstracts and annotations.

Finally, I want to thank the many people who invested their time in reviewing earlier drafts of this module: Mark Borger, Peter Feiler, Robert Goldberg, John Goodenough, Marc Graham, John Nestor, K. C. Tai, Christian Wild, and Steven Zeil. Their many commments and suggestions were invaluable in bringing this module to fruition.

Contents

Unit Testing and Analysis

Module Revision History

Version 1.2 (April 1989)	Outline error and other minor errors corrected
Version 1.1 (December 1988)	Minor changes and more thorough annotations in bibliography Approved for publication
Version 1.0 (October 1987)	Draft for public review

Unit Testing and Analysis

Capsule Description

This module examines the techniques, assessment, and management of unit testing and analysis. Testing and analysis strategies are categorized according to whether their coverage goal is functional, structural, error-oriented, or a combination of these. Mastery of the material in this module allows the software engineer to define, conduct, and evaluate unit tests and analyses and to assess new techniques proposed in the literature.

Philosophy

Program testing and analysis are the most practiced means of verifying that a program possesses the features required by its specification. *Testing* is a dynamic approach to *verification* in which code is executed with *test data* to assess the presence (or absence) of required features. *Analysis* is a static approach to verification in which required features are detected by analyzing, but not executing, the code. Many analysis techniques have become established technologies with their own substantial literature. So that they may be given adequate treatment elsewhere, these techniques have been placed outside the scope of this module. Included among these techniques are proof of correctness, safety analysis, and the more open-ended analysis procedures represented by code inspections and reviews.

This module focuses on *unit*-level verification; integration and systems verification are to be covered elsewhere. What constitutes a "unit" has been left imprecise—it may be as little as a single statement or as much as a set of coupled subroutines. The essential characteristic of a unit is that it can meaningfully be treated as a whole. Some of the techniques presented here require associated documentation that states the desired features of the unit. This documentation may be a comment in the source program, a specification written in a formal language, or a general statement of requirements. Unless otherwise indicated, this documentation should not be assumed to be the particular document in the software life cycle called a "software specification," "software requirements definition," or the like. Any document containing information about the unit may provide useful information for testing or analysis.

Three major classes of testing and analysis are discussed—functional, structural, and error-oriented—as well as some hybrid approaches. Functional testing and analysis ensure that specified major features of the software are *covered.* Structural testing and analysis ensure that major characteristics of the code are covered. Error-oriented testing and analysis ensure that the range of typical errors is covered. The potential benefits of each major class are complementary, and no single technique is comprehensive. By specifying the criteria that must be satisfied by a test, each technique acts both as specifier and evaluator—as specifier by indicating features that must be satisfied by the test data, and as evaluator by indicating deficiencies in the test data. Exploring this dual role of test criteria is an important facet of this module.

Assessment of unit testing and analysis techniques can be theoretical or empirical. This module presents both of these forms of assessment, and discusses criteria for selecting methods and controlling the verification process.

Management of unit testing and analysis should be systematic. It proceeds in two stages. First, techniques appropriate to the project must be selected. Then these techniques must be systematically applied.

Objectives

The following is a list of possible educational objectives based upon the material in this module. Objectives for any particular unit of instruction may be drawn from these or related objectives, as may be appropriate to audience and circumstances.

Knowledge

- Define the basic terminology of testing and analysis (particularly those terms found in the glossary or italicized in the text).
- State the theoretical and computational limitations on testing.
- State the strengths and weaknesses of several testing and analysis techniques.

Comprehension

- Explain the complementary nature of functional, structural, and error-oriented testing and analysis techniques.
- Describe how the choice of testing and analysis criteria affects the selection and evaluation of test data.
- Explain the role of error collection as a feedback and control mechanism.

Application

- Test a software unit using functional, structural, and error-oriented techniques.
- Use configuration management to control the process of unit testing and analysis.

Analysis

- Determine the unit testing and analysis techniques applicable to a project, based upon the verification goals, the nature of the product, and the nature of the testing environment.

Synthesis

- Write a test plan tailored to accommodate project constraints.
- Design software tools to support structural testing and analysis techniques.

Evaluation

- Evaluate the potential usefulness of new unit testing and analysis techniques proposed in the literature.

Prerequisite Knowledge

For the functional testing component, the student should be able to read algebraic, axiomatic, and functional specifications of software modules. The structural testing component requires knowledge of BNF grammars and graphs. If structural analysis tools are to be built, the student needs to have knowledge of parsing technology, parse trees, and graph algorithms at the level of an introductory compiler construction course. To understand the fundamental limitations of testing, the student should be familiar with the halting problem and reduction proofs. If underlying foundations of statistical testing are to be explored in depth, then a full year of statistics is a prerequisite. Effective use of the statistical models requires one semester of statistics.

Module Content

Every effort has been made to adhere to the terminology in [IEEE83a] and [IEEE87]. The definition of "testing" found in [IEEE83a], however, has been rejected in favor of distinct definitions for "testing" and "analysis." These and other significant terms used in this module can be found in the glossary, which follows the annotated outline.

Outline

I. Preliminaries

II. Functional Testing and Analysis
  1. Functional analysis
  2. Functional testing
    a. Testing independent of the specification technique
    b. Testing dependent on the specification technique

III. Structural Testing and Analysis
  1. Structural analysis
    a. Complexity measures
    b. Data flow analysis
    c. Symbolic execution
  2. Structural Testing
    a. Statement testing
    b. Branch testing
    c. Conditional testing
    d. Expression testing
    e. Path testing

IV. Error-Oriented Testing and Analysis
  1. Statistical methods
  2. Error-based testing
    a. Fault estimation
    b. Domain testing
    c. Perturbation testing
  3. Fault-based testing
    a. Local extent, finite breadth
    b. Global extent, finite breadth
    c. Local extent, infinite breadth
    d. Global extent, infinite breadth

V. Hybrid Testing Techniques
  1. Structural/functional
  2. Structural/error-based
  3. Functional/error-based

VI. Evaluation of Unit Testing and Analysis Techniques
  1. Theoretical
  2. Empirical

VII. Managerial Aspects of Unit Testing and Analysis
  1. Selecting techniques
    a. Goals
    b. Nature of the product
    c. Nature of the testing environment
  2. Control
    a. Configuration control
    b. Conducting tests

Annotated Outline

I. Preliminaries

As with any emerging field, terminology used in program testing is far from being fixed. There are two standards, [IEEE83a] and [IEEE87], that do a credible job of clarifying confusing, yet commonly used, terms. Every effort has been made in this module to adhere to this terminology. An important variance, however, is the use of the the two key terms in the title—*testing* and *analysis*. *Testing* is taken here to refer only to execution-based evaluation of software. It includes the activities of test data selection, program execution and data collection, and evaluation of results. *Analysis* is taken to refer to non–execution-based evaluation of software. In [IEEE83a], testing is given a broader definition, encompassing the notions described by both "testing" and "analysis." (See *Glossary*, page 12.)

Some problems cannot be solved on a computer because they are either intractible or undecidable. An *intractible* problem is one whose best known solution requires an inordinate amount of resources. An *undecidable* problem is one for which no algorithmic solution is possible in a programming language. There are many such intractible and undecidable problems associated with testing and analysis. In general, programs cannot be exhaustively tested (tested for each input) because to do so is both intractible and undecidable. Huang shows that to test exhaustively a program that reads two 32-bit integers takes on the order of 50 billion years [Huang75]! Even if the input space is smaller, on the very first input it may be the case that the program does not halt within a reasonable time. It may even be the case that it is obvious the correct output will be produced if the program ever halts. The exhaustive test can only be completed, therefore, if all non-halting cases can be detected and eliminated. The problem of effecting such detection, however, is undecidable.

II. Functional Testing and Analysis

1. Functional analysis

Functional analysis seeks to verify, without execution, that the code faithfully implements the specification. Various approaches are possible. In *proof of correctness*, a formal proof is constructed to verify that a program correctly implements its intended function [Berztiss88]. In *safety analysis*, potentially dangerous behavior is identified and steps are taken to ensure such behavior is never manifested [Leveson87].

Functional analysis is mentioned here for completeness, but a discussion of it is outside the scope of this module.

2. Functional testing

Program testing is *functional* when test data are developed from documents that specify a module's intended behavior. These documents include, but are not limited to, the actual specification and the high- and low-level design of the code to be tested [Howden80a, Howden80b]. The goal is to test for each *software feature* of the specified behavior, including the input domains, the output domains, categories of inputs that should receive equivalent processing, and the processing functions themselves.

a. Testing independent of the specification technique

Specifications detail the assumptions that may be made about a given software unit. They must describe the interface through which access to the unit is given, as well as the behavior once such access is given. The *interface* of a unit includes the features of its inputs, its outputs, and their related value spaces (called *domains*). The *behavior* of a module always includes the function(s) to be computed (its *semantics*), and sometimes the *runtime characteristics*, such as its space and time complexity. Functional testing derives test data from the features of the specification.

(i) Testing based on the interface

Testing based on the interface of a module selects test data based on the features of the input and output domains of the module and their interrelationships.

(1) Input domain testing

In *extremal testing*, test data are chosen to cover the extremes of the input domain. Similarly, *midrange testing* selects data from the interiors of domains. The motivation is inductive—it is hoped that conclusions about the entire input domain can be drawn from the behavior elicited by some representative members of it [Myers79]. For structured input domains, combinations of extremal points for each component are chosen. This procedure can generate a large quantity of data, though considerations of the inherent relationships among components can ameliorate this problem somewhat [Howden80a].

(2) Equivalence partitioning

Specifications frequently partition the set of all possible inputs into classes that receive equivalent treatment. Such partitioning is called *equivalence partitioning* [Myers79]. A result of equivalence partitioning is the identification of a finite set of functions and their associated input and output domains. For example, the specification

$$\{(x,y) \mid x \geq 0 \supset y=x \ \& \ x<0 \supset y=-x\}$$

partitions the input into two sets, associated, respectively, with the identity and negation functions. Input constraints and error conditions can also result from this partitioning. Once these partitions have been developed, both extremal and midrange testing are applicable to the resulting input domains.

(3) Syntax checking

Every robust program must parse its input and handle incorrectly formatted data. Verifying this feature is called *syntax checking*. One means of accomplishing this is to execute the program using a broad spectrum of test data. By describing the data with a BNF grammar, instances of the input language can be generated using algorithms from automata theory. [Duncan81] and [Bazzichi-82] describe systems that provide limited control over the data to be generated.

(ii) Testing based on the function to be computed

Equivalence partitioning results in the identification of a finite set of functions and their associated input and output domains. Test data can be developed based on the known *characteristics* of these functions. Consider, for example, a function to be computed that has fixed points, *i.e.*, certain of its input values are mapped into themselves by the function. Testing the computation at these fixed points is possible, even in the absence of a complete specification [Weyuker82]. Knowledge of the function is essential in order to ensure adequate coverage of the output domains.

(1) Special value testing

Selecting test data on the basis of features of the function to be computed is called *special value testing* [Howden80c]. This procedure is particularly applicable to mathematical computations. Properties of the function to be computed can aid in selecting points which will indicate the accuracy of the computed solution. For example, the periodicity of the sine function suggests use of test data values which differ by multiples of 2π. Such characteristics are not unique to mathematical computations, of course. Most prettyprinters, for example, when applied to their own output, should reproduce it unchanged. Some word processors behave this way as well.

(2) Output domain coverage

For each function determined by equivalence partitioning, there is an associated output domain. *Output domain coverage* is performed by selecting points that will cause the extremes of each of the output domains to be achieved [Howden80a]. This ensures that modules have been checked for maximum and minimum output conditions and that all categories of error messages have, if possible, been produced. In general, constructing such test data requires knowledge of the function to be computed and, hence, expertise in the application area.

b. Testing dependent on the specification technique

The specification technique employed can aid in testing. An executable specification can be used as an *oracle* and, in some cases, as a test generator. Structural properties of a specification can guide the testing process. If the specification falls within certain limited classes, properties of those classes can guide the selection of test data. Much work remains to be done in this area of testing.

(i) Algebraic

In algebraic specification, properties of a data abstraction are expressed by means of axioms or rewrite rules. In one testing system, DAISTS, the consistency of an algebraic specification with an implementation is checked by testing [Gannon81]. Each axiom is compiled into a procedure, which is then associated with a set of test points. A driver program supplies each of these points to the procedure of its respective axiom. The procedure, in turn, indicates whether the axiom is satisfied. Structural coverage of both the implementation and the specification is computed. [Jalote83] discusses another approach in which axioms are used to generate test data.

(ii) Axiomatic

Despite the potential for widespread use of predicate calculus as a specification language, little has been published about deriving test data from such specifications. [Gourlay81] explores the relationship between predicate calculus specifications and path testing.

(iii) State machines

Many programs can be specified as state machines, thus providing an additional means of selecting test data [Beizer83]. Since the equivalence problem of two finite automata is decidable, testing can be used to decide whether a program that simulates a finite automaton with a bounded number of nodes is equivalent to the one specified. This result can be used to test those features of programs that can be specified by finite automata, *e.g.*, the control flow of a transaction-processing system.

(iv) Decision tables

Decision tables are a concise method of representing an equivalence partitioning. The rows of a decision table specify all the conditions that the input may satisfy. The columns specify different sets of actions that may occur. Entries in the table indicate whether the actions should be performed if a condition is satisfied. Typical entries are "Yes," "No," or "Don't Care." Each row of the table suggests significant test data. *Cause-effect graphs* [Myers79] provide a systematic means of translating English specifications into decision tables, from which test data can be generated.

III. Structural Testing and Analysis

In structural program testing and analysis, test data are developed or evaluated from the source code [Howden75]. The goal is to ensure that various characteristics of the program are adequately covered.

1. Structural analysis

In *structural analysis*, programs are analyzed without being executed. The techniques resemble those used in compiler construction. The goal here is to identify fault-prone code, to discover anomalous circumstances, and to generate test data to cover specific characteristics of the program's structure.

a. Complexity measures

As resources available for testing are always limited, it is necessary to allocate these resources efficiently. It is intuitively appealing to suggest that the more complex the code, the more thoroughly it should be tested. Evidence from large projects seems to indicate that a small percentage of the code typically contains the largest number of errors. Various complexity measures have been proposed, investigated, and analyzed in the literature. [McCabe83] contains several pertinent articles, as well as references to others.

b. Data flow analysis

A program can be represented as a flow graph annotated with information about *variable definitions*, *references*, and *undefinitions*. From this representation, information about *data flow* can be deduced for use in code optimization, anomaly detection, and test data generation [Hecht77, Muchnick81]. Data flow *anomalies* are flow conditions that deserve further investigation, as they may indicate problems. Examples include: defining a variable twice with no intervening reference, referencing a variable that is undefined, and undefining a variable that has not been referenced since its last definition. Algorithms for detecting these anomalies are given in [Fosdick76] and [Osterweil76] and are refined and corrected in [Jachner84]. Data flow analysis can also be used in test data generation, exploiting the relationship between points where variables are defined and points where they are used [Rapps85, Laski83, Ntafos84].

c. Symbolic execution

A *symbolic execution system* accepts three inputs: a program to be interpreted, *symbolic input* for the program, and the path to follow. It produces two outputs: the *symbolic output* that describes the computation of the selected path, and the *path condition* for that path. The specification of the path can be either interactive [Clarke76] or preselected [Howden77, Howden78b]. The symbolic output can be used to prove the program correct with respect to its specification, and the path condition can be used for generating test data to exercise the desired path. Structured data types cause difficulties, however, since it is sometimes impossible to deduce what component is being modified in the presence of symbolic values [Hantler76].

2. Structural Testing

Structural testing is a *dynamic* technique in which test data selection and evaluation are driven by the goal of covering various characteristics of the code during testing [Howden75, Huang75, Myers79]. Assessing such coverage involves instrumenting the code to keep track of which characteristics of the program text are actually exercised during testing. The inexpensive cost of such instrumentation has been a prime motivation for adopting this technique [Probert82]. More importantly, structural testing addresses the fact that only the program text reveals the detailed decisions of the programmer. For example, for the sake of efficiency, a programmer might choose to implement a special case that appears nowhere in the specification. The corresponding code will be tested only by chance using functional testing, whereas use of a structural coverage measure such as statement coverage should indicate the need for test data for this case. Structural coverage measures form a rough hierarchy, with higher levels being more costly to perform and analyze, but being more beneficial, as described below.

a. Statement testing

Statement testing requires that every statement in the program be executed. While it is obvious that achieving 100% statement coverage does not ensure a correct program, it is equally obvious that anything less means that there is code in the program that has never been executed!

b. Branch testing

Achieving 100% statement coverage does not ensure that each branch in the program flow graph has been executed. For example, executing an **if** ... **then** statement (no else) when the tested condition is true, tests only one of two branches in the flow graph. *Branch testing* seeks to ensure that *every* branch has been executed [Huang75]. Branch coverage can be checked by probes inserted at points in the program that represent arcs from branch points in the flow graph [Probert82]. This instrumentation suffices for statement coverage as well.

c. Conditional testing

In *conditional testing*, each clause in every condition is forced to take on each of its possible values in combination with those of other clauses

[Huang75]. Conditional testing thus subsumes branch testing and, therefore, inherits the same problems as branch testing. Instrumentation for conditional testing can be accomplished by breaking compound conditional statements into simple conditions and nesting the resulting if statements.

d. Expression testing

Expression testing [Hamlet77a] requires that every expression assume a variety of values during a test in such a way that no expression can be replaced by a simpler expression and still pass the test. If one assumes that every statement contains an expression and that conditional expressions form a proper subset of all the program expressions, then this form of testing properly subsumes all the previously mentioned techniques. Expression testing does require significant runtime support for the instrumentation [Hamlet77b].

e. Path testing

In *path testing*, data are selected to ensure that all paths of the program have been executed. In practice, of course, such coverage is impossible to achieve, for a variety of reasons. First, any program with an indefinite loop contains an infinite number of paths, one for each iteration of the loop. Thus, no finite set of data will execute all paths. The second difficulty is the infeasible path problem: it is undecidable whether an arbitrary path in an arbitrary program is executable. Attempting to generate data for such infeasible paths is futile, but it cannot be avoided. Third, it is undecidable whether an arbitrary program will halt for an arbitrary input. It is therefore impossible to decide whether a path is finite for a given input.

In response to these difficulties, several simplifying approaches have been proposed. Infinitely many paths can be partitioned into a finite set of equivalence classes based on characteristics of the loops. *Boundary and interior testing* require executing loops zero times, one time, and, if possible, the maximum number of times [Howden75]. *Linear sequence code and jump criteria* [Woodward80] specify a hierarchy of successively more complex path coverage. [Howden78a], [Tai80], [Gourlay83], and [Weyuker86] suggest methods of studying the adequacy of structural testing.

Path coverage does not imply condition coverage or expression coverage since an expression may appear on multiple paths but some subexpressions may never assume more than one value. For example, in

$$\textbf{if } a \vee b \textbf{ then } S_1 \textbf{ else } S_2$$

b may be false and yet each path may still be executed.

IV. Error-Oriented Testing and Analysis

Testing is necessitated by the potential presence of errors in the programming process. Techniques that focus on assessing the presence or absence of errors in the programming process are called *error-oriented*. There are three broad categories of such techniques: *statistical assessment*, *error-based testing*, and *fault-based testing*. These are stated in order of increasing specificity of what is wrong with the program. Statistical methods attempt to estimate the *failure rate* of the program without reference to the number of remaining faults.

Error-based testing attempts to show the absence of certain errors in the programming process. Fault-based testing attempts to show the absence of certain faults in the code. Since errors in the programming process are reflected as faults in the code, both techniques demonstrate the absence of faults. They differ, however, in their starting point: error-based testing begins with the programming process, identifies potential errors in that process and then asks how those errors are reflected as faults. It then seeks to demonstrate the absence of those reflected faults. Fault-based testing begins with the code and asks what are the potential faults in it, regardless of what error in the programming process caused them.

1. Statistical methods

Statistical testing employs statistical techniques to determine the *operational reliability* of the program. Its primary concern is how *faults* in the program affect its failure rate in its actual operating environment. A program is subjected to test data that statistically model the operating environment, and failure data are collected. From these data, a reliability estimate of the program's failure rate is computed. [Currit86] explains this method for use in an incremental development environment and cites other relevant sources. A statistical method for testing paths that compute algebraic functions is given in [DeMillo78b]. There has been a prevailing sentiment that statistical testing is a futile activity, since it is not directed toward finding errors [DeMillo78a, Myers79]. However, studies suggest it is a viable alternative to structural testing [Duran80, Duran84]. Combining statistical testing with an oracle appears to represent an effective tradeoff of computer resources for human time [Panzl81].

2. Error-based testing

Error-based testing seeks to demonstrate that certain *errors* have not been committed in the programming process [Weyuker83]. Error-based testing can be driven by histories of programmer errors, measures of software complexity, knowledge of error-prone syntactic constructs, or even error guessing [Myers79]. Some of the more methodical techniques are described below.

a. Fault estimation

Fault seeding is a statistical method used to assess the number and characteristics of the faults remaining in a program. A reprint of Harlan Mills' original proposal for this technique (where he calls it *error seeding*) appears in [Mills83]. First, faults are seeded into a program. Then the program is tested, and the number of faults discovered is used to estimate the number of faults yet undiscovered. A difficulty with this technique is that the faults seeded must be representative of the yet-undiscovered faults in the program.

Techniques for predicting the quantity of remaining faults can also be based on a reliability model. A survey of reliability models is found in [Ramamoorthy82].

b. Domain testing

The input domain of a program can be partitioned according to which inputs cause each path to be executed. These partitions are called *path domains*. Faults that cause an input to be associated with the wrong path domain are called *domain faults*. Other faults are called *computation faults*. (The terms used before attempts were made to rationalize nomenclature were "domain errors" and "computation errors.") The goal of *domain testing* is to discover domain faults by ensuring that test data limit the range of undetected faults [White80]. [Clarke82] refines the fault detection capability of this approach.

c. Perturbation testing

Perturbation testing attempts to decide what constitutes a sufficient set of paths to test. Faults are modeled as a vector space, and characterization theorems describe when sufficient paths have been tested to discover both computation and domain errors. Additional paths need not be tested if they cannot reduce the dimensionality of the error space [Zeil83].

3. Fault-based testing

Fault-based testing aims at demonstrating that certain prescribed faults are not in the code. It functions well in the role of test data evaluation: test data that do not succeed in discovering the prescribed faults are not considered adequate. Fault-based testing methods differ in both *extent* and *breadth*. One with local extent demonstrates that a fault has a local effect on computation; it is possible that this local effect will not produce a program *failure*. A method with global extent demonstrates that a fault *will* cause a program failure. Breadth is determined by whether the technique handles a finite or an infinite class of faults. Extent and breadth are orthogonal, as evidenced by the techniques described below.

a. Local extent, finite breadth

In [Hamlet77a] and [Hamlet77b], a system built into a compiler to judge the adequacy of test data is described. Input-output pairs of data are encoded as a comment in a procedure, as a partial specification of the function to be computed by that procedure. The procedure is then executed for each of the input values and checked for the output values. The test is considered adequate only if each computational or logical expression in the procedure is *determined* by the test; *i.e.*, no expression can be replaced by a simpler expression and still pass the test. *Simpler* is defined in a way that allows only finitely many substitutions. Thus, as the procedure is executed, each possible substitution is evaluated on the data state presented to the expression. Those that do not evaluate the same as the original expression are rejected. The system allows methods of specifying the extent to be analyzed.

b. Global extent, finite breadth

In *mutation testing*, test data adequacy is judged by demonstrating that interjected faults are caught. A program with interjected faults is called a *mutant*, and is produced by applying a *mutation operator*. Such an operator changes a single expression in the program to another expression, selected from a finite class of expressions. For example, a constant might be incremented by one, decremented by one, or replaced by zero, yielding one of three mutants. Applying the mutation operators at each point in a program where they are applicable forms a finite, albeit large, set of mutants. The test data are judged adequate only if each mutant in this set is either functionally equivalent to the original program or computes different output than the original program. Inadequacy of the test data implies that certain faults can be introduced into the code and go undetected by the test data.

Mutation testing is based on two hypotheses. The *competent programmer hypothesis* says that a competent programmer will write code that is close to being correct; the correct program, if not the current one, can be produced by some straightforward syntactic changes to the code. The *coupling effect hypothesis* says that test data that reveal simple faults will uncover complex faults as well. Thus, only single mutants need be eliminated, and combinatoric effects of multiple mutants need not be considered [DeMillo78a]. [Gourlay83] formally characterizes the competent programmer hypothesis as a function of the probability of the test set's being reliable (as defined by Gourlay) and shows that under this characterization, the hypothesis does not hold. Empirical justification of the coupling effect has been attempted [DeMillo78a, Budd80], but theoretical analysis has shown that it is does not hold, even for simple programs [Morell83].

c. Local extent, infinite breadth

[Foster80] describes a method for selecting test data that are sensitive to errors. Howden has formalized this approach in a method called *weak mutation testing* [Howden82]. Rules for recognizing error-sensitive data are described for each primitive language construct. Satisfaction of a rule for a given construct during testing means that all alternate forms of that construct have been distinguished. This has an obvious advantage over mutation testing—elimination of all mutants without generating a single one! Some rules even allow for infinitely many mutants. Of course, since this method is of local extent, some of the mutants eliminated may indeed be the correct program. [Morell83] extends this method for global extent.

d. Global extent, infinite breadth

[Morell83] and [Morell87] define a fault-based method based on symbolic execution that permits elimination of infinitely many faults through evidence of global failures. *Symbolic faults* are inserted into the code, which is then executed on real or symbolic data. Program output is then an expression in terms of the symbolic faults. It thus reflects how a fault at a given location will impact the program's output. This expression can be used to determine actual faults that could not have been substituted for the symbolic fault and remain undetected by the test.

V. Hybrid Testing Techniques

Since it is apparent that no one testing technique is sufficient, some people have investigated ways of integrating several techniques. Such integrated techniques are called *hybrid testing techniques*. These are not just the concurrent application of distinct techniques; they are characterized by a deliberate attempt to incorporate the best features of different methods into a new framework.

1. Structural/functional

In *partition analysis*, test data are chosen to ensure simultaneous coverage of both the specification and code [Richardson85]. An operational specification language has been designed that enables a structural measure of coverage of the specification. The input space is partitioned into a set of domains that is formed by the cross product of path domains of the specification and path domains of the program. Test data are selected from each non-empty partition, ensuring simultaneous coverage of both specification and code. Proof of correctness techniques can also be applied to these cross product domains. The testing system DAISTS predates and automates this technique for algebraic specifications of abstract data types, but it does not include any notion of proof of correctness [Gannon81]. Furthermore, the emphasis in DAISTS is on test data evaluation, rather than generation. [Goodenough75] presents a less formal, integrated scheme for selecting test data based on analysis of sources of errors in the programming process.

2. Structural/error-based

This area has concentrated on *data flow faults* in which values computed at one location in the program are misused at another location. Discovering these faults requires information derived from data flow analysis of the program [Laski83, Ntafos84, Rapps85]. A promising area of research is the use of models of program faults to guide test data selection. Code complexity measures can be used to identify segments of the program to be subjected to intensive testing.

3. Functional/error-based

Research in this area has been slow because error-based data applicable to functional testing is difficult to collect, difficult to assess, and frequently proprietary [Glass81, Ostrand84]. It is possible that fault seeding combined with statistical testing may prove to be a useful method.

VI. Evaluation of Unit Testing and Analysis Techniques

The effectiveness of unit testing and analysis may be evaluated on theoretical or empirical grounds [Howden78a]. Theory seeks to understand what can be done *in principle*; empirical evaluation seeks to establish what techniques are useful *in practice*. Theory formally defines the field and investigates its fundamental limitations. For example, it is well known that testing cannot demonstrate the correctness of an arbitrary program with respect to an arbitrary specification. This does not mean, however, that testing can never verify correctness; indeed, in some cases it can [Howden78c]. Empirical studies evaluate the utility of various practices. While statement testing is theoretically deficient, it is immensely useful in practice, catching many program faults.

IEEE has sponsored four workshops on testing between 1978 and 1988. Proceedings are published for only the most recent of the three [WST88]. The National Bureau of Standards has issued a special publication that describes most of the techniques mentioned in this module and characterizes each approach according to effectiveness, applicability, learning, and cost [NBS82].

1. Theoretical

Theory serves three fundamental purposes: to define terminology, to characterize existing practice, and to suggest new avenues of exploration. Unfortunately, current terminology is inconsistent. A simple example is the word *reliable*, which is used by authors in related, but varying ways. (Compare, for example, [Goodenough75], [Howden76], [Duran81], [Hamlet81], and [Richardson85], and do not include any authors from *reliability theory*!) [IEEE83a] is a good starting point for examining terminology, but it is imprecise in places and was established many years after certain (in retrospect, unfortunate) terminology had become accepted.[1] Theoretical treatments of topics in program testing are few. Goodenough and Gerhart, in [Goodenough75], made an attempt to rationalize terminology, though this work has been criticized, particularly in [Weyuker80]. Nevertheless, they anticipated the vast majority of practical and theoretical issues that have since evolved in program testing. [Goodenough75] is therefore required reading. Howden and Weyuker have both written theoretical expositions on functional and structural testing [Howden76, Howden78a, Howden78c, Howden82, Howden86, Weyuker80, Weyuker82, Weyuker83, Rapps85, Weyuker84, Weyuker86]. Theoretical expositions of mutation and fault-based testing are found in [Hamlet77a], [Hamlet81], [Budd82], [Gourlay81], [Gourlay83], and [Morell83].

2. Empirical

Empirical studies provide benchmarks by which to judge existing testing techniques. An excellent comparison of techniques is found in [Howden80b], which emphasizes the complementary benefits of structural and functional testing for scientific programs. Empirical studies of mutation testing are discussed in [Budd80]. Many articles that report experience with various testing techniques appear in the proceedings of the ACM Symposium on Principles of Programming Languages, the International Conference on Software Engineering, and the Computer Software and Applications Conference.

VII. Managerial Aspects of Unit Testing and Analysis

Administration of unit testing and analysis proceeds in two stages. First, techniques appropriate to the project must be selected. Then these techniques must be systematically applied. [IEEE87] provides explicit guidance for these steps.

1. Selecting techniques

Selecting the appropriate techniques from the array of possibilities is a complex task that requires assessment of many issues, including the goal of testing, the nature of the software product, and the nature of the test environment. It is important to remember the complementary benefits of the various techniques and to select as broad a range of techniques as possible, within imposed limits. No single testing or analysis technique is sufficient. Functional testing suffers from inadequate code coverage, structural testing suffers from inadequate specification coverage, and neither technique achieves the benefits of error coverage.

a. Goals

Different design goals impose different demands on the selection of testing techniques. Achieving correctness requires use of a great variety of techniques. A goal of reliability implies the need for statistical testing using test data representative of those of the anticipated user environment. It should be noted, however, that proponents of this technique still recommend judicious use of "selective" tests to avoid embarrassing or disastrous situations [Currit86]. Testing may also be directed toward assessing the utility of proposed software. This kind of testing requires a solid foundation in human factors [Shneiderman79, Perlman88]. Performance of the software may also be of special concern. In this case, extremal testing is essential. Timing instrumentation can prove useful.

Often, several of these goals must be achieved simultaneously. One approach to testing under these circumstances is to order testing by decreasing benefit. For example, if reliability, correctness, and performance are all desired features, it is reasonable to tackle performance first, reliability second, and correctness third, since these goals require increasingly difficult-to-design tests. This approach can have the beneficial effect of identifying faulty code with less effort expended.

b. Nature of the product

The nature of the software product plays an important role in the selection of appropriate techniques. Four types of software products are discussed below.

(i) Data processing

Data processing applications appear to benefit from most of the techniques described in this module. Conventional languages such as COBOL are frequently used, increasing the likelihood of finding an instrumented compiler for doing performance and coverage analysis. Functional test cases are typically easy to identify. Even domain testing, with its many restrictions, seems applicable, since most predicates in data processing programs are linear [White80].

[1] For instance, the terminology ***error-based testing*** and ***error seeding*** became well-established long before the standard told us to use *fault*.

(ii) Scientific computation

Howden analyzed a variety of static and dynamic testing and analysis techniques on the IMSL routines [Howden80b]. He concluded that functional and structural testing are complementary, that neither is sufficient, and that sometimes a hybrid approach is necessary to simultaneously cover extremal values while executing a particular path. Static methods found fewer errors in Howden's study, but their earlier application in the life cycle may increase their effectiveness.

Extremal value testing and special value testing are vital to scientific programs. Statistical testing is perhaps less appropriate, since these programs are frequently constructed to solve problems whose characteristics are not known in advance. The IMSL package illustrates this; the designers of the package cannot make "reasonable" assumptions about the distribution of arguments to the *sine* routine, for instance.

(iii) Expert systems

Expert systems pose unique challenges to verification. Structural testing is of little use, since the behavior of the system is dominated by the knowledge base. Difficulties arise in assuring the consistency of this knowledge base. This problem is compounded by the reliance on human experts, since precise behavior is difficult to specify. A good survey of the problems related to validation of expert systems appears in [Hayes-Roth83].

Three steps can be identified as minimal requirements for verification of an expert system. First, it is necessary to clean up the knowledge base in much the same manner as is done for a BNF grammar. Inconsistencies must be detected, redundancies eliminated, loops broken, etc. Symbolic execution and data flow analysis appear to be applicable to this stage. Second, each piece of information in the knowledge base must be exercised. Mutation analysis applied to the knowledge base detects the information whose change does not affect the output and, thus, is not sufficiently exercised. Third, test case design and evaluation must be conducted by experts in the application domain.

(iv) Embedded and real-time systems

Embedded and real-time systems are perhaps the most complex systems to specify, design, and implement. It is no surprise that they are particularly hard to verify. Embedded computer systems typically have inconvenient interfaces for defining and conducting tests. Ultimately, the code must execute on the embedded computer in its operational environment. *Operational testing* is performed in this environment.

Unit testing in an operational environment is rarely possible. The equipment is seldom available and may lack conventional input and output. In these cases, the embedded computer can be placed in a controlled environment that simulates the operational one. This provides the capability of conducting a *system test*. Timing constraints must be verified here. To assess time-critical software, it is essential to collect data in as unobtrusive a manner as possible. Typically, this requires hardware instrumentation, though software breakpoints sometimes suffice. Data from several points of instrumentation must be coordinated and analyzed; such a process is called *data reduction*.

If the simulated environment does not support unit testing, the embedded computer itself must be abstracted. The software can be written in assembly language, and the embedded computer can be simulated on another machine; or the software can be written in a high-level language, such as Ada, which can be cross-compiled to the target machine. At this level of abstraction, unit testing and analysis are possible. The goal during this stage is to assess correctness of individual units. Functional testing is essential, especially extremal, midrange, and special value testing, since it is impossible to ensure these tests will occur during integration or system testing. Data flow analysis of the code, especially if the system is written in assembly language, is appropriate. A simulator can be instrumented to collect necessary code coverage statistics.

c. Nature of the testing environment

Available resources, personnel, and project constraints must be considered in selecting testing and analysis strategies.

(i) Available resources

Available resources frequently determine the extent of testing. If the compiler does not instrument code, if data flow analysis tools are not at hand, if exotic tools for mutation testing or symbolic evaluation are not available, one must perform functional testing and hand instrument the code to detect branch coverage. Hand instrumentation is not difficult, but it is an error-prone and time consuming process. Editor scripts can aid in the instrumentation process. If resources permit, successively more complex criteria involving branch testing, data flow testing, domain testing, and fault-based testing can be tried.

(ii) Personnel

No technique is without its personnel costs. Before introducing any new technique or tool, the impact on personnel must be considered. The advantages of any approach must be balanced against the effort required to learn the technique, the ongoing time demands of applying it, and the expertise it requires. Domain testing can be quite difficult to learn. Data flow analysis may uncover many anomalies that are not errors, thereby requiring personnel to sort through and distinguish them. Special value testing requires expertise in the application area. Analysis of the impact on personnel for many of the techniques in this module can be found in [NBS82].

(iii) Project constraints

The goal in selecting testing and analysis techniques is to obtain the most benefit from testing within the project constraints. Testing is indeed over when the budget or the time allotted to it is exhausted, but this is not an appropriate definition of when to stop testing [Myers79]. Estimates indicate that approximately 40% of software development time is used in the testing phase. Scheduling must reflect this fact.

2. Control

To ensure quality in unit testing and analysis, it is necessary to control both documentation and the conduct of the test.

a. Configuration control

Several items from unit testing and analysis should be placed under configuration management, including the test plan, test procedures, test data, and test results. A formal description of these and related items is found in [IEEE83b]. The test plan specifies the goals, environment, and constraints imposed on testing. The test procedures detail the step-by-step activities to be performed during the test. *Regression testing* occurs when previously saved test data are used to test modified code. Its principal advantage is that it ensures previously attained functionality has not been lost during a modification. Test results are recorded and analyzed for evidence of program failures. Failure rates underly many reliability models [Ramamoorthy82]; high failure rates may indicate the need for redesign.

b. Conducting tests

A *test bed* is an integrated system for testing software. Several such systems exist, for example, AUT [Panzl78], RXVP [Miller74], TPL/F [Panzl-78], ITB [Miller81a], LDRA [Hennell83, Woodward80], IVTS [Senn83], and, to a limited extent, SimplT [Hamlet77b] and DAISTS [Gannon81]. Minimally, these systems provide the ability to define a test case, construct a test driver, execute the test case, and capture the output. Additional facilities provided by such systems typically include data flow analysis, structural coverage assessment, regression testing, test specification, and report generation.

Glossary

The following terminology is used throughout the module, except possibly in the abstracts in the bibliography. Additional terms are defined in the text. Note that the literature is replete with inconsistencies in the use of such terms as "error," "failure," and "fault." Consistent use of these terms has been attempted here, but such consistency may itself lead to confusion in the many cases where "modern" usage conflicts with usage prevalent in the literature.

analysis
The process of evaluating without execution a system or system component to verify that it satisfies specified requirements. (*Cf.* definition of "testing" in [IEEE83a].)

coverage
Used in conjunction with a software feature or characteristic, the degree to which that feature or characteristic is tested or analyzed. Examples include input domain coverage, statement coverage, branch coverage, and path coverage.

error
Human action that results in software containing a fault [IEEE83a].

failure
The inability of a system or system component to perform a required function within specified limits [IEEE83a].

fault
A manifestation of an error in software [IEEE83a].

oracle	A mechanized procedure that decides whether a given input-output pair is acceptable.
software characteristic	An inherent, possibly accidental, trait, quality, or property of software [IEEE87].
software feature	A software characteristic specified or implied by requirements documentation [IEEE87].
test bed	A test environment containing the hardware, instrumentation tools, simulators, and other support software necessary for testing a system or a system component [IEEE83a].
test data	Data developed to test a system or system component [IEEE83a].
testing	The process of executing a system or system component on selected test data to verify that it satisfies specified requirements. (*Cf.* definition of "testing" in [IEEE83a].)
unit	Code that is meaningful to treat as a whole. It may be as small as a single statement or as large as a set of coupled subroutines.
verification	The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase [IEEE83a].

Teaching Considerations

Textbooks

There are currently three books on the subject of testing and analysis. [Myers79] is the least comprehensive, but it provides a good overview of structural coverage and some functional testing. It can still serve well as a supplementary text in an introductory software engineering course. [Beizer83] is eclectic, containing more dynamic testing techniques than any other reference. It does not cover static methods, such as data flow analysis and symbolic execution. The text is written in a style that will captivate the student's attention, however, and it references many actual projects. [Howden87] is the first text to approach testing and analysis from a unified framework. It contains the necessary theoretical and practical background and could be used as a text for a graduate seminar.

For a full appreciation of issues, each of these texts will have to be supplemented by readings from the current literature. *Suggested Reading Lists* (page 15) contains a table categorizing the bibliography entries according to potential use.

Suggested Schedules

The following are suggestions for using the material in this module. The parenthesized numbers represent approximate number of lecture hours allocated for each topic.

One-Term Undergraduate Introduction to Software Engineering. The large quantity of material to be covered in this course makes it difficult to deal with any topic in depth. The following minimum coverage of unit testing and analysis issues is suggested:

- Theory (0.5)
- Functional Testing and Analysis (1.5)
- Structural Testing and Analysis (1.5)
- Managerial Aspects (1.5)

Total: 5.0 hours

Undergraduate Course on Verification Techniques. A course covering proof of correctness, review techniques, and testing and analysis provides a springboard for understanding the complicated issues in verification.

Suggested coverage:

- Theory (1.0)
- Functional Testing and Analysis (4.0)
- Structural Testing and Analysis (4.0)
- Error-based Testing and Analysis (3.0)
- Managerial Aspects (3.0)

Total: 15 hours

Graduate Seminar on Testing and Analysis. As indicated in the table in *Suggested Reading Lists*, there is a wealth of material to support a graduate seminar in testing. The entire outline of this module can be covered, with additional topics included as deemed appropriate. The suggestions given below focus on how this material can be taught in a seminar format.

The instructor delivers an introductory lecture in each of the major topic areas. Lectures should be based on papers from the "essential" category (column 1 of the table). A subset of papers from the "recommended" list (column 2) is selected to be read by all students; one student should act as presenter for each paper. For this approach to succeed, papers and presenters must be selected well in advance, and both presenters and participants must be prepared. To ensure this advanced preparation, the instructor should:

- Approve all paper selections.
- Meet with each presenter at least two weeks in advance of the presentation to answer questions, determine presentation format, and together write a set of exercises for the other students.
- Distribute the assigned reading as soon as possible and the set of exercises at least one week in advance of the presentation.
- Be prepared to assist each student at his presentation, if necessary.

This approach requires discipline on everyone's part.

Broad coverage of material is aided by requiring each student to write a term paper in one of the areas related to the course. Readings listed in the "essential" and "recommended" columns provide breadth, while those categorized as "detailed" or "expert" provide depth.

Suggested coverage:

- Background and Theory (5.0)
- Functional Testing and Analysis
    - Testing independent of the specification technique (6.0)
    - Testing dependent on the specification technique (3.0)
- Structural Testing and Analysis
    - Static analysis (6.0)
    - Testing (3.0)
- Error-Oriented Testing and Analysis
    - Statistical methods (2.0)
    - Error-based testing (4.0)
    - Fault-based testing (3.0)
- Hybrid Testing Techniques (3.0)
- Evaluation of Unit Testing and Analysis
    - Theoretical (3.0)
    - Empirical (1.0)
- Managerial Aspects of Unit Testing and Analysis
    - Selecting techniques (1.0)
    - Configuration items (1.0)
    - Testbeds (1.0)

Total: 42 hours

Exercises

It is not sufficient merely to study techniques—they must be applied to software and evaluated. Fortunately there is no lack of software to be verified! The traditional projected-oriented software engineering course clearly should have a testing component. If a testing seminar is held concurrently with such a course, the students taking the seminar can act as an independent test organization, as tool builders, as consultants, etc. for the software engineering class. Alternatively, programs can be obtained from another class or from industry for sustained testing.

In a testing seminar, the complementary benefits of functional and structural testing can be illustrated by dividing the seminar participants into two groups. Have each group produce a specification and fault-filled program. The functional group receives the specification and object code from the structural group, which, in turn, receives only the source code from the functional group. After testing is complete, the groups compare results. Roles of the two groups can then be reversed.

Testing tools are prime candidates for projects. Rudimentary testbeds, data flow analyzers, and code instrumenters can be implemented in one term. Tools developed during one term can serve both as the test tools and the test objects for the next.

Suggested Reading Lists

The following lists categorize items in the bibliography by applicability. "Essential" reading is exactly that. "Recommended" reading provides additional background. "Detailed" reading indicates papers with narrow scope. These papers can serve as the basis for a class project. "Expert" reading is for those who have a good background in a particular area; this category contains mostly theoretical papers. Coverage of the first two categories, including one detailed area, is about all that can be expected in a single term.

Paper Categories

Essential	Recommended	Detailed	Expert
Clarke82	Duran81	Bazzichi82	Budd82
Currit86	Foster80	Budd80	DeMillo78b
DeMillo78a	Gannon81	Clarke76	Gourlay81
Duran84	Glass81	Duncan81	Gourlay83
Fosdick76	Hamlet77b	Duran80	Hamlet81
Gerhart76	Howden78a	Hamlet77a	Howden78c
Goodenough75	Howden80a	Hennell83	Morell83
Hantler76	Howden82	Howden75	Morell87
Howden76	Huang75	Howden77	Ramamoorthy82
Howden80b	Jachner84	Howden78b	Weyuker83
Howden80c	Kemmerer85	Jalote83	Weyuker84
Howden86	Laski83	Miller74	Weyuker86
IEEE83a	NBS82	Miller81b	
IEEE83b	Ostrand84	Ntafos84	
IEEE87	Panzl78	Osterweil76	
Richardson85	Probert82	Panzl81	
Weyuker80	Rapps85	Redwine83	
White80	Tai80	Roussopoulos85	
	Weyuker82	Rowland81	
	Woodward80	Senn83	
	Zeil83	Weiser84	
		White85	

Bibliography

The following annotations include comments about the value of each work to the instructor and student. Works identified as "recommended reading" or "should be read" provide necessary background reading. Items labeled "could prove useful" are narrow in scope and of limited applicability. Papers described as "for experts only" should be reserved for student punishment.

Several of the following articles are reprinted in the tutorial by Miller and Howden [Miller81a]; they are so noted.

Bazzichi82

Bazzichi, Franco, and Ippolito Spadafora. "An Automatic Generator for Compiler Testing." *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 343-353.

Abstract: A new method for testing compilers is presented. The compiler is exercised by compatible programs, automatically generated by a test generator. The generator is driven by a tabular description of the source language. This description is in a formalism which nicely extends context-free grammars in a context-dependent direction, but still retains the structure and readability of BNF. The generator produces a set of programs which cover all grammatical constructions of the source language, unless user supplied directives instruct otherwise. The programs generated can also be used to evaluate the performance of different compilers of the same source language.

A significant example from Pascal is presented, and experience with the generator is reported.

The approach taken here is one similar to that of a two-level grammar for specifying context sensitivity. The problems inherent in specifying semantic constraints on a programming language are clearly presented. However, the presentation is difficult to understand without consulting the references in the bibliography.

This paper or [Duncan81] should be read by the instructor. It is a difficult paper for students, though its goal should be apparent.

Beizer83

Beizer, Boris. *Software Testing Techniques*. New York: Van Nostrand, 1983.

Table of Contents

1. Introduction
2. The Taxonomy of Bugs
3. Flowcharts and Path Testing
4. Path Testing and Transaction Flows
5. Graphs, Paths and Complexity
6. Paths, Path Products, and Regular Expressions
7. Data Validation and Syntax Testing
8. Data-Base-Driven Test Design
9. Decision Tables and Boolean Algebra
10. Boolean Algebra the Easy Way
11. States, State Graphs, and Transition Testing
12. Graph Matrices and Applications

Berztiss88

Berztiss, Alfs and Mark A. Ardis. *Formal Verification of Programs*. Curriculum Module SEI-CM-20-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.

Budd80

Budd, Timothy A., Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Theoretical and Empirical Studies on Using Program Mutations to Test the Functional Correctness of Programs." *Conf. Record 7th Annual ACM Symposium on Principles of Prog. Lang.* New York: ACM, Jan. 1980, 220-233.

This paper presents little-known results on mutation testing, including both theoretical and empirical studies. The theoretical section can be safely ignored, except for the analysis of decision tables and straight-line Lisp programs. The empirical results are more interesting, since they provide insight into the mutant operators used and their success on buggy programs.

The theoretical section is useful only for those who wish to pursue mutation testing at an expert level. The empirical section is of some use in demonstrating when mutation testing does and does not work.

Budd82

Budd, Timothy A., and Dana Angluin. "Two Notions of Correctness and Their Relation to Testing." *Acta Informatica 18*, 1 (1982), 31-45.

Abstract: We consider two interpretations for what it means for test data to demonstrate correctness. For each interpretation, we examine under what conditions data sufficient to demonstrate correctness exists, and whether it can be automatically detected and/or generated. We establish the relation between these questions and the problem of deciding equivalence of two programs.

This article requires a good background in computability theory. A theoretical analysis of mutation testing is presented in excellent style.

This paper is for experts. Students without a course in computability will be lost.

Clarke76

Clarke, Lori A. "A System to Generate Test Data and Symbolically Execute Programs." *IEEE Trans. Software Eng. SE-2*, 3 (Sept. 1976), 215-222.

Abstract: This paper describes a system that attempts to generate test data for programs written in ANSI Fortran. Given a path, the system symbolically executes the path and creates a set of constraints on the program's input variables. If the set of constraints is linear, linear programming techniques are employed to obtain a solution. A solution to the set of constraints is test data that will drive execution down the given path. If it can be determined that the set of constraints is inconsistent, then the given path is shown to be nonexecutable. To increase the chance of detecting some of the more common programming errors, artificial constraints are temporarily created that simulate error conditions and then an attempt is made to solve each augmented set of constraints. A symbolic representation of the program's output variables in terms of the program's input variables is also created. The symbolic representation is in a human readable form that facilitates error detection as well as being a possible aid in assertion generation and automatic program documentation.

Clarke82

Clarke, Lori A., Johnette Hassell, and Debra J. Richardson. "A Close Look at Domain Testing." *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 380-390.

Abstract: White and Cohen have proposed the domain testing method, which attempts to uncover errors in a path domain by selecting test data on and near the boundary of the path domain. The goal of domain testing is to demonstrate that the boundary is correct within an acceptable error bound. Domain testing is intuitively appealing in that it provides a method for satisfying the often suggested guideline that boundary conditions should be tested.

In addition to proposing the domain testing method, White and Cohen have developed a test data selection strategy, which attempts to satisfy this method. Further, they have described two error measures for evaluating domain testing strategies. This paper takes a close look at their strategy and their proposed error measures. It is shown that inordinately large domain errors may remain undetected by the White and Cohen strategy. Two alternative domain testing strategies, which improve on the error bound, are then proposed and the complexity of each of the three strategies is analyzed. Finally, several other issues that must be addressed by domain testing are presented and the general applicability of this method is discussed.

This paper recommends the selection of additional test points to narrow the range of domain shifts that remain undetected by the domain testing strategy suggested in [White80], which is essential prerequisite reading. The paper makes several important suggestions for relaxing the restrictions of [White80].

This is essential reading for the instructor if domain testing is to be discussed and also serves as a good source of thought questions for examinations. It is advanced reading for students.

Currit86

Currit, P. A., Michael Dyer, and Harlan D. Mills. "Certifying the Reliability of Software." *IEEE Trans. Software Eng. SE-12*, 1 (Jan. 1986), 3-11.

Abstract: The accepted approach to software development is to specify and design a product in response to a requirements analysis and then to test the software selectively with cases perceived to be typical to those requirements. Frequently the result is a product which works well against inputs similar to those tested but which is unreliable in unexpected circumstances.

In contrast it is possible to embed the software development and testing process within a formal statistical design. In such a design, software testing can be used to make statistical inferences about the reliability of the future operation of the software. In turn, the process of systematically assessing reliability permits a certification of the product at delivery, that attests to a public record of defect detection and repair and to a measured level of operating reliability.

This paper describes a procedure for certifying the reliability of software before its release to users. The ingredients of this procedure are a life cycle of executable product increments, representative statistical testing, and a standard estimate of the MTTF (mean time to failure) of the product at the time of its release.

The paper discusses the development of certified software products and the derivation of a statistical model used for reliability projection. Available software test data are used to demonstrate the application of the model in the certification process.

This paper provides a model of software reliability based upon incremental software development. Failure rates are determined from operational testing of each incremental release. The focus is on eliminating faults that have the greatest impact on operational performance.

A strong background in statistics is necessary for full appreciation of the mathematics. The paper is essential reading for the instructor.

DeMillo78a

DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer 11*, 4 (April 1978). Reprinted in [Miller81a].

This paper should win a prize for introducing more sexy new terms than any other—*mutation testing, competent programmer hypothesis, coupling effect.* Beware! It is easy to fall under the spell of the latter two terms and assume they are well-defined and justified. Beware also of the typographical error that occurs in several places on page 37, where '1' is substituted for 'I'. This substitution leads to the (wrong) impression that mutation testing is performed conventionally on double mutants. The description on page 39 is confusing and seems to imply that 14 mutants are equivalent to the original, yet four of them are not. [Duran81] draws exactly the opposite conclusion based on the random generation example!

Despite flaws, this paper is a marvelous introduction to mutation testing and is, therefore, essential reading for both instructor and student.

DeMillo78b

DeMillo, Richard A., and Richard J. Lipton. "A Probabilistic Remark on Algebraic Program Testing." *Information Processing Letters 7*, 4 (June 1978), 193-195.

Duncan81

Duncan, A. G., and J. S. Hutchinson. "Using Attributed Grammars to Test Designs and Implementations." *5th Intl. Conf. on Software Eng.* New York: IEEE, March 1981, 170-178.

Abstract: We present a method for generating test cases that can be used throughout the entire life cycle of a program. This method uses attributed translation grammars to generate both inputs and outputs, which can then be used either as is, in order to test the specifications, or in conjunction with automatic test drivers to test an implementation against the specifications.

The grammar can generate test cases either randomly or systematically. The attributes are used to guide the generation process, thereby avoiding the generation of many superfluous test cases. The grammar itself not only drives the generation of test cases but also serves as a concise documentation of the test plan.

In the paper, we describe the test case generator, show how it works in typical examples, compare it with related techniques, and discuss how it can be used in conjunction with various testing heuristics.

This is a practical paper on the means of generating test data based on a BNF grammar. The use of "attributes" here is unconventional and is not directly related to attribute grammars.

This paper or [Bazzichi82] should be read by the instructor. It can be useful for in-depth study by the student.

Duran80

Duran, Joe W., and John J. Wiorkowski. "Quantifying Software Validity by Sampling." *IEEE Trans. on Reliability R-29*, 2 (June 1980), 141-144.

Abstract: The point of all validation techniques is to raise assurance about the program under study, but no current methods can be realistically thought to give 100% assurance that a validated program will perform correctly. There are currently no useful ways for quantifying how 'well-validated' a program is. One measure of program correctness is the proportion of elements in the program's input domain for which it fails to execute correctly, since the proportion is zero i.f.f. the program is correct. This proportion can be estimated statistically from the results of program tests and from prior subjective assessments of the program's correctness. Three examples are presented of methods for determining s-confidence bounds on the failure proportion. It is shown that there are reasonable conditions (for programs with a finite number of paths) for which ensuring the testing of all paths does not *give better assurance of program correctness.*

The authors are interested in program testing, particularly in quantifying how well a program has been tested. Both random testing and path testing are considered. A strong statistical background is presumed.

Expert reading for the instructor.

Duran81

Duran, Joe W., and John J. Wiorkowski. "Capture-Recapture Sampling for Estimating Software Error Content." *IEEE Trans. Software Eng. SE-7*, 1 (Jan. 1981), 147-148.

Abstract: Mills' capture-recapture sampling method allows the estimation of the number of errors in a program by randomly inserting known errors and then testing the program for both inserted and indigenous errors. This correspondence shows how correct confidence limits and maximum likelihood *estimates can be obtained from the test results. Both fixed sample size testing and sequential testing are considered.*

It is essential that Mills' original article be read first (see chapter 9 of [Mills83]). A strong statistics background is needed.

This reading is for experts only.

Duran84

Duran, Joe W., and Simeon C. Ntafos. "An Evaluation of Random Testing." *IEEE Trans. Software Eng. SE-10*, 4 (July 1984), 438-444.

Abstract: Random testing of programs has usually (but not always) been viewed as a worst case of program testing. Testing strategies that take into account the program structure are generally preferred. Path testing is an often proposed ideal for structural testing. Path testing is treated here as an instance of partition testing, where by partition testing is meant any testing scheme which forces execution of at least one test case from each subset of a partition of the input domain. Simulation results are presented which suggest that random testing may often be more cost effective than partition testing schemes. Also, results of actual random testing experiments are presented which confirm the viability of random testing as a useful validation tool.

This paper challenges many ideas about program testing, especially the notion that random testing is of no value. Experiments were conducted to validate an error model, and the structural coverage accomplished by such testing is reported. Knowledge of statistics helps.

This is essential reading for the instructor. It is challenging for the student, but it should be read.

Fosdick76

Fosdick, Lloyd D., and Leon J. Osterweil. "Data Flow Analysis in Software Reliability." *ACM Computing Surveys 8*, 3 (Sept. 1976), 305-330.

Abstract: The ways that the methods of data flow analysis can be applied to improve software reliability are described. There is also a review of the basic terminology from graph theory and from data flow analysis in global program optimization. The notation of regular expressions is used to describe actions on data for sets of paths. These expressions provide the basis of a classification scheme for data flow which represents patterns of data flow along paths within subprograms and along paths which cross subprogram boundaries. Fast algorithms, originally introduced for global optimization, are described and it is shown how they can be used to implement the classification scheme. It is then shown how these same algorithms can also be used to detect the presence of data flow anomalies which are symptomatic of programming errors. Finally, some characteristics of and experience with DAVE, a data flow analysis system embodying some of these ideas, are described.

This article is a most readable and thorough introduction to data flow analysis. Read this first and compare with [Jachner84].

This is essential reading for both instructor and student.

Foster80

Foster, Kenneth A. "Error Sensitive Test Cases Analysis (ESTCA)." *IEEE Trans. Software Eng. SE-6*, 3 (May 1980), 258-264.

Abstract: A hardware failure analysis technique adapted to software yielded three rules for generating test cases sensitive to code errors. These rules, and a procedure for generating these cases, are given with examples. Areas for further study are recommended.

A set of error-sensitive test case analysis rules are given for producing inputs that are "error-sensitive." The rules are *ad hoc*, and no theoretical justification is given for them. An error in the article is corrected in *Software Engineering Notes 10*, 1 (Jan. 1985), 62-67.

This article contains many classical examples and is useful for that reason. It is not essential reading, but it raises many questions about why the proposed ideas seems to work.

Gannon81

Gannon, John, Paul R. McMullin, and Richard G. Hamlet. "Data-Abstraction, Implementation, Specification, and Testing." *ACM Trans. Prog. Lang. and Syst. 3*, 3 (July 1981), 211-223.

Abstract: A compiler-based system DAISTS that combines a data-abstraction language (derived from the SIMULA class) with specification by algebraic axioms is described. The compiler, presented with two independent syntactic objects in the axioms and implementing code, compiles a "program" that consists of the former as test driver for the latter. Data points, in the form of expressions using the abstract functions and constant values, are fed to this program to determine if the implementation and axioms agree. Along the way, structural testing measures can be applied to both code and axioms to evaluate the test data. Although a successful test does not conclusively demonstrate the consistency of axioms and code, in practice the tests are seldom successful, revealing errors. The advantage over conventional programming systems is threefold:

(1) The presence of the axioms eliminates the need for a test oracle; only inputs need be supplied

(2) Testing is automated: a user writes axioms, implementation, and test points; the system writes the test drivers.

(3) the results of tests are often surprising and helpful because it is difficult to get away with "trivial" tests: what is not significant for the code is liable to be a severe test of the axioms, and vice versa.

The system described here covers many diverse aspects of program testing. It is a specification-dependent hybrid approach that takes advantage of the orthogonality between implementations and algebraic axioms.

This paper is recommended reading for the instructor. With some background in algebraic specification, students can readily comprehend the system.

Gerhart76

Gerhart, Susan L., and Lawrence Yelowitz. "Observations of Fallibility in Applications of Modern Programming Methodologies." *IEEE Trans. Software Eng. SE-2*, 3 (Sept. 1976), 195-207.

Abstract: Errors, inconsistencies, or confusing points are noted in a variety of published algorithms, many of which are being used as examples in formulating or teaching principles of such modern programming methodologies as formal specification, systematic construction, and correctness proving. Common properties of these points of contention are abstracted. These properties are then used to pinpoint possible causes of the errors and to formulate general guidelines which might help to avoid further errors. The common characteristic of mathematical rigor and reasoning in these examples is noted, leading to some discussion about fallibility in mathematics, and its relationship to fallibility in these programming methodologies. The overriding goal is to cast a more realistic perspective on the methodologies, particularly constructive recommendations for their improvement.

This paper is a masterpiece of analysis of how errors occur in the life cycle. Though a bit "nit-picky" in places, the paper succeeds in convincing the most adamant skeptic of the need for dynamic testing of computer programs. It is best understood after some formal specifications and proofs of correctness are attempted.

This paper is essential reading for both instructor and student.

Glass81

Glass, Robert L. "Persistent Software Errors." *IEEE Trans. Software Eng. SE-7*, 2 (March 1981), 162-168.

Abstract: Persistent software errors—those which are not discovered until late in development, such as when the software becomes operational—are by far the most expensive kind of error. Via analysis of software problem reports, it is discovered that the predominant number of persistent errors in large-scale software efforts are errors of omitted logic..., that is, the code is not as complex as required by the problem to be solved. Peer design and code review, desk checking, and ultra-rigorous testing may be the most helpful of the currently available technologies in attacking this problem. New and better methodologies are needed.

Goodenough75

Goodenough, John B., and Susan L. Gerhart. "Toward a Theory of Test Data Selection." *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 156-173. Reprinted in [Miller81a].

Abstract: This paper examines the theoretical and practical role of testing in software development. We prove a fundamental theorem showing that properly structured tests are capable of demonstrating the absence of errors in a program. The theorem's proof hinges on our definition of test reliability and validity, but its practical utility hinges on being able to show when a test is actually reliable. We explain what makes tests unreliable (for example, we show by example why testing all program statements, predicates, or paths is not usually sufficient to insure test reliability), and we outline a possible approach to developing reliable tests. We also show how the analysis required to define reliable tests can help in checking a program's design and specifications as well as in preventing and detecting implementation errors.

Despite the flaws indicated in [Weyuker80], this remains a classic.

The paper is essential reading for the instructor. Students find it very difficult; do not use it as an introduction to testing!

Gourlay81

Gourlay, John S. *Theory of Testing Computer Programs*. Ph.D. Th., University of Michigan, 1981.

Gourlay83

Gourlay, John S. "A Mathematical Framework for the Investigation of Testing." *IEEE Trans. Software Eng. SE-9*, 6 (Nov. 1983), 686-709.

Abstract: Testing has long been in need of mathematical underpinnings to explain its value as well as its limitations. This paper develops and applies a mathematical framework that 1) unifies previous work on the subject, 2) provides a mechanism for *comparing the power of methods of testing programs based on the degree to which the methods approximate program verification, and 3) provides a reasonable and useful interpretation of the notion that successful tests increase one's confidence in the program's correctness.*

Applications of the framework include confirmation of a number of common assumptions about practical testing methods. Among the assumptions confirmed is the need for generating tests from specifications as well as programs. On the other hand, a careful formal analysis shows that the "competent programmer hypothesis" does not suffice to ensure the claimed high reliability of mutation testing. Hardware testing is shown to fit into the framework as well, and a brief consideration of it shows how the practical differences between it and software testing arise.

This paper is expert reading.

Hamlet77a

Hamlet, Richard G. "Testing Programs with Finite Sets of Data." *Computer Journal 20*, 3 (Aug. 1977), 232-237.

Abstract: The techniques of compiler optimization can be applied to aid a programmer in writing a program which cannot be improved by these techniques. A finite, representative set of test data can be useful in this process. This paper presents the theoretical basis for the (nonconstructive) existence of test sets which serves as maximally effective stand-ins for an unlimited number of input possibilities. It is argued that although the time required by a compiler to fully exercise a program on a set of data may be large, the corresponding improvement in the reliability of the program may also be large if the set meets the given theoretical requirements.

As a theoretical companion to [Hamlet77b], this paper explores the notion of assessing test data adequacy via program mutations. The article requires some background in computability, especially in reduction proofs involving the halting problem.

The paper could be used as an introduction to computability for students with limited background, since all its theorems are relevant to issues involved in program testing.

Hamlet77b

Hamlet, Richard G. "Testing Programs with the Aid of a Compiler." *IEEE Trans. Software Eng. SE-3*, 4 (July 1977), 279-290.

Abstract: If finite input-output specifications are added to the syntax of programs, these specifications can be verified at compile time. Programs which carry adequate tests with them in this way should be resistant to maintenance errors. If the specifications are independent of program details they are easy to give, and unlikely to contain errors in common with the program. Furthermore, certain finite specifications are maximal in that they exercise the control and expression structure of a program as well as any tests can.

A testing system based on a compiler is described, in which compiled code is utilized under interactive control, but "semantic" errors are reported in the style of conventional syntax errors. The implementation is entirely in the high-level language on which the system is based, using some novel ideas for improving documentation without sacrificing efficiency.

This paper provides an excellent description of a system that anticipated many of the fault-based methods of program testing practice and theory. It represents the first fault-based system using program mutation in a context that determines test data adequacy by demonstrating that no simpler programs can be substituted for the original and still pass the test. The paper is easy to understand and motivates discussion of mutation testing and test data adequacy.

This paper is recommended reading for the instructor and provides an interesting comparison of tradeoffs among mutation methods for the student.

Hamlet81

Hamlet, Richard G. "Reliability Theory of Program Testing." *Acta Informatica 16*, 1 (1981), 31-43.

Hantler76

Hantler, Sidney L. and James C. King. "An Introduction to Proving the Correctness of Programs." *ACM Computing Surveys 8*, 3 (Sept. 1976), 331-353. Reprinted in [Miller81a].

Abstract: This paper explains, in an introductory fashion, the method of specifying the correct behavior of a program by the use of input/output assertions and describes one method for showing that the program is correct with respect to those assertions. An initial assertion characterizes conditions expected to be true upon entry to the program and a final assertion characterizes conditions expected to be true upon exit from the program. When a program contains no branches, a technique known as symbolic execution can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit. More generally, for a program with branches one can define a symbolic execution tree. If there is an upper bound on the number of times each loop in such a program may be executed, a proof of correctness can be given by a simple traversal of the (finite) symbolic execution tree.

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding symbolic execution trees are infinite. In order to prove the correctness of such programs, a more general assertion structure must be provided. The symbolic execution tree of such programs must be traversed inductively rather than explicitly. This leads naturally to the use of additional assertions which are called "inductive assertions."

This article provides an excellent introduction to three important areas: program correctness, formal verification, and symbolic execution. It is highly readable and provides a gentle introduction to these areas.

The instructor who needs to learn about the relationship of symbolic execution to verification should begin here, but will need additional background as well. It is ideal for student reading.

Hayes-Roth83

Hayes-Roth, Frederick and Donald Arthur Waterman, eds. *Building Expert Systems.* Reading, Mass.: Addison-Wesley, 1983.

Hecht77

Hecht, Matthew S. *Flow Analysis of Computer Programs.* New York: Elsevier North-Holland, 1977.

This book covers the theory of data flow analysis as applied to program optimization. Application of data flow analysis to verification is not covered.

Hennell83

Hennell, M. A., D. Hedley, and I. J. Riddlell. "The LDRA Software Testbeds: Their Roles and Capabilities." *Proc. SOFTFAIR: A Conf. on Software Development Tools, Techniques, and Alternatives.* New York: IEEE, July 1983, 69-77.

Abstract: The LDRA software Testbeds are described from two distinct viewpoints: those of the individual users of the test beds, and of the managers whose role it is to enforce standards. The discussion looks at the information yielded and considers how this can be used to increase the quality of implemented software.

The paper summarizes how the underlying research work and the accumulated experience of ten years in-service use have been incorporated into the Testbeds.

Both the programmer's view and management's view of a test bed is presented. The system supports a variety of static and dynamic testing methods for several languages. No references to other test beds are given.

As a representative paper on software test beds, this paper deserves reading. Students will be able to understand it.

Howden75

Howden, William E. "Methodology for the Generation of Program Test Data." *IEEE Trans. Computers C-24,* 5 (May 1975), 554-560.

Abstract: A methodology for generating program test data is described. The methodology is a model of the test data generation process and can be used to characterize the basic problems of test data generation. It is well defined and can be used to build an automatic test data generation system.

The methodology decomposes a program into a finite set of classes of paths in such a way that an intuitively complete set of test cases would cause the execution of one path in each class. The test data generation problem is theoretically unsolvable: there is no algorithm which, given any class of paths, will either generate a test case that causes some path in that class to be followed or determine that no such data exist. The methodology attempts to generate test data for as many of the classes of paths as possible. It operates by constructing descriptions of the input data subsets which cause the classes of paths to be followed. It transforms these descriptions into systems of predicates which it attempts to solve.

This paper contains a nuts-and-bolts presentation of symbolic execution techniques.

The instructor may find this paper useful, but it is somewhat dated. Students who are not enthusiastic about using structural coverage to generate test data should avoid this one.

Howden76

Howden, William E. "Reliability of the Path Analysis Testing Strategy." *IEEE Trans. Software Eng. SE-2,* 3 (Sept. 1976), 208-215. Reprinted in [Miller81a].

Abstract: A set of test data T for a program P is reliable *if it reveals that P contains an error whenever P is incorrect. If a set of tests T is reliable and P produces the correct output for each element of T then P is a correct program. Test data generation strategies are procedures for generating sets of test data. A testing strategy is reliable for a program P if it produces a reliable set of test data for P. It is proved that an effective testing strategy which is reliable for all programs cannot be constructed. A description of the path analysis testing strategy is presented. In the path analysis strategy data are generated which cause different paths in a program to be executed. A method for analyzing the reliability of path testing is introduced. The method* *is used to characterize certain classes of programs and program errors for which the path analysis strategy is reliable. Examples of published incorrect programs are included.*

This is an excellent paper that established much of the terminology and influenced much of the work in path testing.

It is essential reading for both the instructor and student.

Howden77

Howden, William E. "Symbolic Testing and the DISSECT Symbolic Evaluation System." *IEEE Trans. Software Eng. SE-3*, 4 (July 1977), 266-278. Reprinted in [Miller81a].

Abstract: Symbolic testing and a symbolic evaluation system called DISSECT are described. The principle features of DISSECT are outlined. The results of two classes of experiments in the use of symbolic evaluation are summarized. Several classes of program errors are defined and the reliability of symbolic testing in finding bugs is related to the classes of errors. The relationship of symbolic evaluation systems like DISSECT to classes of program errors and to other kinds of program testing and program analysis tools is also discussed. Desirable improvements in DISSECT, whose importance was revealed by the experiments, are mentioned.

This paper provides a detailed look into the strengths and weaknesses of a symbolic execution system. Several interesting notions are introduced, such as using two-dimensional output to improve readability of symbolic output, and the use of a path description language.

The paper is necessary only for in-depth understanding of symbolic execution. It is easily understood by students.

Howden78a

Howden, William E. "Theoretical and Empirical Studies of Program Testing." *IEEE Trans. Software Eng. SE-4*, 4 (July 1978), 293-298.

Abstract: Two approaches to the study of program testing are described. One approach is theoretical and the other empirical. In the theoretical approach situations are characterized in which it is possible to use testing to formally prove the correctness of programs or the correctness of properties of programs. In the empirical approach testing strategies reveal the errors in a collection of programs. A summary of the results of two research projects which investigated these approaches are presented. The differences between the two approaches are discussed and their relative advantages and disadvantages are compared.

This paper is recommended reading for the instructor who wishes to compare the theoretical approach with the empirical approach. It is readily understood by students.

Howden78b

Howden, William E. "DISSECT—A Symbolic Evaluation and Program Testing System." *IEEE Trans. Software Eng. SE-4*, 1 (Jan. 1978), 70-73.

Abstract: The basic features of the DISSECT symbolic testing tool are described. Usage procedures are outlined and the special advantages of the tool are summarized. Cost estimates for using the tool are provided and the results of experiments to determine its effectiveness are included. The background and history of the development of the tool are outlined. The availability of the tool is described and a listing of reference materials is included.

Howden78c

Howden, William E. "Algebraic Program Testing." *Acta Informatica 10*, 1 (1978), 53-66.

Abstract: An approach to the study of program testing is introduced in which program testing is treated as a special kind of equivalence problem. In this approach, classes of programs P* *and associated classes of test sets* T* *are defined which have the property that if two programs* P *and* Q *in* P* *agree on a set of tests from* T*, *then* P *and* Q *are computationally equivalent. The properties of a class* P* *and the associated class* T* *can be thought of as defining a set of assumptions about a hypothetical correct version* Q *of a program* P *in* P*. *If the assumptions are valid then it is possible to prove the correctness of* P *by testing. The main result of the paper is an equivalence theorem for classes of programs which carry out sequences of computations involving the elements of arrays.*

This reading is for expert knowledge.

Howden80a

Howden, William E. "Functional Testing and Design Abstractions." *J. Syst. and Software 1*, 4 (April 1980), 307-313. Reprinted in [Miller81a].

Howden80b

Howden, William E. "Applicability of Software Validation Techniques to Scientific Programs." *ACM Trans. Prog. Lang. and Syst.* 2, 3 (July 1980), 307-320. Reprinted in [Miller81a].

Abstract: Error analysis involves the examination of a collection of programs whose errors are known. Each error is analyzed and validation techniques which would discover the error are identi-

fied. The errors that were present in version five of a package of Fortran scientific subroutines and then later corrected in version six were analyzed. An integrated collection of static and dynamic analysis methods would have discovered the error in version five before its release. An integrated approach to validation and the effectiveness of individual methods are discussed.

An excellent description of what errors are discovered by what techniques.

This paper is essential reading for the instructor and student alike.

Howden80c

Howden, William E. "Functional Program Testing." *IEEE Trans. Software Eng. SE-6*, 2 (March 1980), 162-169.

Abstract: An approach to functional testing is described in which the design of a program is viewed as an integrated collection of functions. The selection of test data depends on the functions used in the design and on the value spaces over which the functions are defined. The basic ideas on the method were developed during the study of a collection of scientific programs containing errors. The method was the most reliable testing technique for discovering the errors. It was found to be significantly more reliable than structural testing. The two techniques are compared and their relative advantages and limitations are discussed.

By functional program testing, Howden means testing those aspects of a program that have any form of external specification, including design documents or even comments within the code. The use of the term *functional testing* in this module is derived from this and similar papers.

[Howden80c] is essential reading for both the instructor and the student.

Howden82

Howden, William E. "Weak Mutation Testing and Completeness of Test Sets." *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 371-379. Reprinted in [Miller81a].

Abstract: Different approaches to the generation of test data are described. Error-based *approaches depend on the definition of classes of commonly occurring program errors. They generate tests which are specifically designed to determine if particular classes of errors occur in a program. An error-based method called* weak mutation testing *is described. In this method, tests are constructed which are guaranteed to force program statements which contain certain classes of errors to act incorrectly during the execution of the program over those tests. The method is systematic, and a tool can be built to help the user apply the method. It is extensible in the sense that it can be extended to cover additional classes of errors. Its relationship to other software testing methods is discussed. Examples are included.*

Different approaches to testing involve different concepts of the adequacy or completeness *of a set of tests. A formalism for characterizing the completeness of test sets that are generated by error-based methods such as weak mutation testing as well as the test sets generated by other testing methods is introduced. Error-based, functional, and structural testing emphasize different approaches to the test data generation problem. The formalism which is introduced in the paper can be used to describe their common basis and their differences.*

As a variant of mutation testing, weak mutation testing is a viable option and bears close resemblance to the system described in [Hamlet77a]. This paper formalizes the notion of completeness of a test set based on its ability to detect local changes to the code. A good comparison of testing methods is made, using the notation introduced in the paper. The paper is more easily understood if [DeMillo78], [White80], and [Foster80] are read first.

This paper is recommended for the instructor, especially if error-based or fault-based testing is to be covered in depth. Given sufficient background, students should find the paper accessible. It could form the basis of a class project to develop a weak mutation system.

Howden86

Howden, William E. "A Functional Approach to Program Testing and Analysis." *IEEE Trans. Software Eng. SE-12*, 10 (Oct. 1986), 997-1005.

Abstract: An integrated approach to testing is described which includes both static and dynamic analysis methods and which is based on theoretical results that prove both its effectiveness and efficiency. Programs are viewed as consisting of collections of functions that are joined together using elementary functional forms or complex functional structures.

Functional testing is identified as the input-output analysis of functional forms. Classes of faults are defined for these forms and results presented which prove the fault revealing effectiveness of well defined sets of tests.

Functional analysis is identified as the analysis of the sequences of operators, functions, and data type transformations which occur in functional structures. Functional trace analysis involves the examination of the sequences of function calls which occur in a program path; operator sequence anal-

ysis the examination of the sequences of operators on variables, data structures, and devices; and data type transformation analysis the examination of the sequences of transformations on data types. Theoretical results are presented which prove that it is only necessary to look at interfaces between pairs of operators and data type transformations in order to detect the presence of operator or data type sequencing errors. The results depend on the definition of normal forms for operator and data type sequencing diagrams.

This paper represents the culmination of Howden's ideas on program testing as a full-blown theory. The article summarizes his book, [Howden87], and should be consulted before selecting the book for a course. By an interesting twist of terminology, Howden has managed to incorporate all of structural testing into functional testing. He presumes the availability of external functions that specify the behavior of components of the program, even those as small as an expression. Thus, conventional structural issues such as branch testing are converted into questions like "Does this condition compute this (externally defined) function?" Of course, the existence of these external functions for every line of code can be questioned, but Howden has a quick reply—you can use the code to generate the function! While such sleight-of-hand may be disturbing at first, it is clear that in some cases this is not inappropriate, as when a section of code fits a standard paradigm and has a comment that says "sort list." To understand fully his development, it is necessary to see Howden's progress through several papers, especially [Howden76], [Howden80c], and [Howden82].

This paper is essential reading for the instructor. The presentation is at such a high level that it will be difficult for an uninitiated student to understand, even though it is very well written.

Howden87

Howden, William E. *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

Table of Contents

1. Introduction
2. Functions
3. States and Types
4. Theoretical Foundations
5. Functional Program Testing
6. Functional Analysis
7. Management and Planning

Huang75

Huang, J. C. "An Approach to Program Testing." *ACM Computing Surveys 8*, 3 (Sept. 1975), 113-128. Reprinted in [Miller81a].

Abstract: One of the practical methods commonly used to detect the presence of errors in a computer program is to test it for a set of test cases. The probability of discovering errors through testing can be increased by selecting test cases in such a way that each and every branch in the flowchart will be traversed at least once during the test. This tutorial describes the problems involved and the methods that can be used to satisfy the test requirement.

This paper discusses a method for determining path conditions to enable branch coverage.

The paper is very easy to understand and should cause no problems for students. It will introduce them to predicate calculus notation for expressing path conditions. It is recommended reading for both instructor and students.

IEEE83a

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. New York: IEEE, 1983. ANSI/IEEE Std 729-1983.

IEEE83b

IEEE. *IEEE Standard for Software Test Documentation*. New York: IEEE, 1983. ANSI/IEEE Std 829-1983.

IEEE87

IEEE. *IEEE Standard for Software Unit Testing*. New York: IEEE, 1987. ANSI/IEEE Std 1008-1987.

Jachner84

Jachner, Jacek, and Vinod K. Agarwal. "Data Flow Anomaly Detection." *IEEE Trans. Software Eng. SE-10*, 4 (July 1984), 432-437.

Abstract: The occurrence of a data flow anomaly is often an indication of the existence of a programming error. The detection of such anomalies can be used for detecting errors and to upgrade software quality. This paper introduces a new, efficient algorithm capable of detecting anomalous data flow patterns in a program represented by a graph. The algorithm based on static analysis scans the paths entering and leaving each node of the graph to reveal anomalous data action combinations. An algorithm implementing this type of approach was proposed by Fosdick and Osterweil [2]. Our approach presents a general framework which not only fills a gap in the previous algorithm, but also provides time and space improvements.

This paper corrects a problem in [Fosdick76] and cannot be understood without having read its predecessor.

If [Fosdick76] is covered, this paper must be read by the instructor. The paper opens up the possibility of a meta-discussion about the need to analyze papers critically. The shock effect on students of the reliability of published papers is not to be underestimated. See [Weyuker80] for additional support for this approach.

Jalote83
Jalote, Pankaj. "Specification and Testing of Abstract Data Types." *Proceedings of COMPSAC 83*. Silver Spring, Md.: IEEE Computer Society Press, Nov. 1983, 508-511.

Abstract: Software testing assumes the existence of a test oracle who is frequently human. The test system described obviates the need for a conventional oracle by automatically generating a test oracle for an Abstract Data Type from its specifications. In certain circumstances the test system may also detect Abstract Data Types that are not completely specified and help diagnose missing axioms. A test point generator is provided which, when used together with this oracle, facilitates the testing of Abstract Data Types.

This article describes a method for testing abstract data types specified by algebraic axioms, in which test data are generated automatically.

The paper provides a detailed example of testing based on the specification technique. It is not essential reading.

Kemmerer85
Kemmerer, Richard A. "Testing Formal Specifications to Detect Design Errors." *IEEE Trans. Software Eng. SE-11*, 1 (Jan. 1985), 32-43.

Abstract: Formal specification and verification techniques are now used to increase the reliability of software systems. However, these approaches sometimes result in specifying systems that cannot be realized or that are not usable. This paper demonstrates why it is necessary to test specifications early in the software life cycle to guarantee a system that meets its critical requirements and that also provides the desired functionality. Definitions to provide the framework for classifying the validity of a functional requirement with respect to a formal specification are also introduced. Finally, the design of two tools for testing formal specifications is discussed.

Laski83
Laski, Janusz W., and Bogdan Korel. "A Data Flow Oriented Program Testing Strategy." *IEEE Trans. Software Eng. SE-9*, 3 (May 1983), 347-354.

Abstract: Some properties of a program data flow can be used to guide program testing. The presented approach aims to exercise use-definition chains that appear in a program. Two such data oriented testing strategies are proposed; the first involves checking liveness of every definition of a variable at the point(s) of its possible use; the second deals with liveness of vectors of variables treated as arguments to an instruction or program block. Reliability of these strategies is discussed with respect to a program containing an error.

This paper provides a transition from the use of data flow to detect anomalies in programs to its use as a method for test data selection and evaluation.

The paper should be read by the instructor and is accessible to students. The difference between using a criterion as an evaluation method rather than as a generation method needs to be emphasized.

Leveson87
Leveson, Nancy. *Software Safety*. Curriculum Module SEI-CM-6-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.

McCabe83
Structured Testing. McCabe, Thomas J., ed. Silver Spring, Md.: IEEE Computer Society Press, 1983.

Miller74
Miller, Edward, Michael R. Paige, Jeoffrey P. Benson, and William R. Wisehart. "Structural Techniques of Program Validation." *Digest of Papers, COMPCON Spring 74*. Northridge, Calif.: IEEE Computer Society, 1974, 161-164. Reprinted in [Miller81a].

Abstract: A structural basis for the formulation of test cases for given computer programs has been found to be an effective and efficient strategy. An existing automated program validation system employs these techniques with good success in minimizing the number of test cases required; this same system permits automatic identification of test cases in a high proportion of instances. Research aimed at fully automating the test case generation process continues.

Level-*i* path testing is defined and illustrated. The presentation is terse and difficult to follow, but it illustrates the analysis performed by one of the earlier testbeds, RXVP.

This is in-depth reading on path testing for the instructor. Student readers will struggle as much with presentation as with content.

Miller81a
Tutorial: Software Testing & Validation Techniques. Miller, Edward and William E. Howden, eds. New York: IEEE Computer Society Press, 1981.

Miller81b

Miller, Edward. "A Software Test Bed: Philosophy, Implementation and Application." *Computer Program Testing: Proc. of the Summer School on Computer Program Testing Held at SOGESTA, Urbino, Italy, June 29-July 3, 1981.* New York: Elsevier North-Holland, 1981, 231-240.

Abstract: This paper outlines some general considerations leading to the development of an integrated automated test system for computer software. An example of the finished system, called ITB, is given.

Mills83

Mills, Harlan D. *Software Productivity.* Boston: Little, Brown, 1983.

Morell83

Morell, Larry J. *A Theory of Error-Based Testing.* Ph.D. Th., University of Maryland, College Park, Md., 1983.

This is available from the author. (See address at the front of this module.)

Morell87

Morell, Larry J. "A Model for Assessing Code-based Testing Techniques." *5th Annual Pacific Northwest Software Quality Conf.* Portland, Oreg.: Lawrence & Craig, Oct. 1987, 309-325. To appear in *IEEE Trans. Software Eng.*

Abstract: A theory of fault-based program testing is defined and explained. Testing is fault-based *when it seeks to demonstrate that prescribed faults are not in a program. It is assumed here that a program can only be incorrect in a limited fashion specified by associating* alternate expressions *with program expressions. Classes of alternate expressions can be infinite. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of fault-based testing is to produce a test set that differentiates the program from each of its alternates.*

A particular form of fault-based testing based on symbolic execution is presented. In symbolic testing *program expressions are replaced by symbolic alternatives that represent classses of alternate expressions. The output from the system is an expression in terms of the input of the symbolic alternative. Equating this with the output from the original program yields a* propagation equation *whose solutions determine those alternatives which are not differentiated by this test.*

Muchnick81

Muchnick, Steven S., and Neil D. Jones, eds. *Program Flow Analysis: Theory and Applications.* Englewood Cliffs, N. J.: Prentice-Hall, 1981.

This book delves deeply into the subject of data flow analysis and many areas of application to testing, including static analysis tools and symbolic execution.

This book is for experts.

Myers79

Myers, Glenford J. *The Art of Software Testing.* New York: John Wiley, 1979.

Table of Contents

1. *A Self-Assessment Test*
2. *Program Inspections, Walkthroughs, and Reviews*
3. *Test-Case Design*
4. *Module Testing*
5. *Higher-Order Testing*
6. *Debugging*
7. *Test Tools and Other Techniques*

This book is an often cited reference on software testing. Although it is somewhat dated, students find it helpful and easy to read.

NBS82

Powell, Patricia B., ed. *Software Validation, Verification, and Testing Technique and Tool Reference Guide.* Washington, D. C.: National Bureau of Standards, 1982.

This book covers most of the testing and analysis techniques covered in this module. The techniques are compared according to their effectiveness, applicability, ease of learning, and costs. The assessments are accurate and succinct.

This is recommended reading for the instructor, since it contains many useful classroom examples.

Ntafos84

Ntafos, Simeon C. "On Required Element Testing." *IEEE Trans. Software Eng. SE-10,* 6 (Nov. 1984), 795-803.

Abstract: In this paper we introduce two classes of program testing strategies that consist of specifying a set of required elements for the program and then covering those elements with appropriate test inputs. In general, a required element has a structural and a functional component and is covered by a test case if the test case causes the features specified in the structural component to be executed under the conditions specified in the functional component. Data flow analysis is used to specify the

structural component and data flow interactions are used as a basis for developing the functional component. The strategies are illustrated with examples and some experimental evaluations of their effectiveness are presented.

A general framework is established for integrating structural testing with data flow information. Reading [Rapps85] first is suggested, which is more comprehensive in treatment of approaches.

The paper could be useful to the instructor, but it is less accessible to the student.

Osterweil76

Osterweil, Leon J., and Lloyd D. Fosdick. "DAVE—A Validation Error Detection and Documentation System for Fortran Programs." *Software—Practice and Experience 6*, 4 (Oct.-Dec. 1976), 473-486. Reprinted in [Miller81a].

Abstract: This paper describes DAVE, a system for analyzing Fortran programs. DAVE is capable of detecting the symptoms of a wide variety of errors in programs, as well as assuring the absence of these errors. In addition, DAVE exposes and documents subtle data relations and flows within programs. The central analytic procedure used is a depth first search. DAVE itself is written in Fortran. Its implementation at the University of Colorado and some early experience is described.

After an abrupt introduction to data flow anomalies, the paper gives two algorithms for computing the input/output classification of a variable. The relationship between these algorithms and the detection of data flow anomalies is not immediately obvious. [Fosdick76] should be read first and correlated with this article. The algorithms are couched in an Algol-like language, making them more palatable to students.

The paper could serve as detailed reading for the instructor. The density of notation makes it difficult for the student.

Ostrand84

Ostrand, Thomas, and Elaine J. Weyuker. "Collecting and Categorizing Software Error Data in an Industrial Environment." *J. Syst. and Software 4*, 11 (Nov. 1984), 289-300.

Abstract: A study has been made of the software errors committed during development of an interactive special-purpose editor system. This product, developed for commercial production use, has been followed during nine months of coding, unit testing, function testing, and system testing. Detected problems and their fixes have been described by testers and debuggers. A new fault categorization scheme was developed from these descriptions and used to classify the 173 faults that resulted from the project's errors. For each error, we asked the programmers to select its most likely cause, report the stages of the software development cycle in which the error was committed and the problem first noticed, and the circumstances of the problem's detection and isolation, including time required, techniques tried, and successful techniques. The results collected in this study are compared to results from earlier studies, and similarities and differences are noted.

Panzl78

Panzl, David J. "Automatic Software Test Drivers." *Computer 11*, 4 (April 1978), 44-50.

This paper is a clear presentation of a method for automating unit tests. It is entirely pragmatic.

[Panzl78] provides many insights into unit testing in an industrial environment and should be read by the instructor. The paper opens up an area of practice to which most students have had little exposure. It also provides a good starting point for potential projects.

Panzl81

Panzl, D. J. "Experience with Automatic Program Testing." *Proc. Trends and Applications 1981: Advances in Software Technology*. New York: IEEE, May 1981, 25-28.

Abstract: A novel method for automating software testing has been demonstrated at the General Electric Corporate Research and Development Center. This method, called the Dual programming method, is based on a new type of automatic software test driver that automates the generation of test case inputs and the checking of test results and thereby allows the execution of tens of thousands of test cases at zero labor cost. Prototype versions of the Dual automatic test driver were implemented on the CRD H6000 computer in 1980. A controlled experiment has shown that the Dual programming method is capable of improving software reliability in individual program modules by one to two orders of magnitude.

Perlman88

Perlman, Gary. *User Interface Development*. Curriculum Module SEI-CM-17-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.

Probert82

Probert, Robert L. "Optimal Insertion of Software Probes in Well-Delimited Programs." *IEEE Trans. Software Eng. SE-8*, 1 (Jan. 1982), 34-42.

Abstract: A standard technique for monitoring software testing activities is to instrument the module under test with counters or probes before testing begins; then, during testing, data generated by these probes can be used to identify portions of as yet unexercised code. In this paper the effect of the disciplined use of language features for explicitly delimiting control flow constructs is investigated with respect to the corresponding ease of software instrumentation. In particular, assuming all control constructs are explicitly delimited, for example, by END IF or equivalent statements, an easily programmed method is given for inserting a minimum number of probes for monitoring statement and branch execution counts without disrupting source code structure or paragraphing. The use of these probes, called statement probes, is contrasted with the use of standard (branch) probes for execution monitoring. It is observed that the results apply to well-delimited modules written in a wide variety of programming languages, in particular, Ada.

Program instrumentation techniques are surveyed, and a specific method is described. The paper is self-contained, and the method described is applicable to most modern languages. A background in graph theory and formal grammars is necessary.

The paper should be read by the instructor if instrumentation is discussed. The paper is explicit enough to form the basis of a class project.

Ramamoorthy82

Ramamoorthy, C. V., and Farokh B. Bastani. "Software Reliability—Status and Perspectives." *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 354-371.

Abstract: It is essential to assess the reliability of digital computer systems used for critical real-time control applications (e.g., nuclear power plant safety control systems). This involves the assessment of the design correctness of the combined hardware/software system as well as the reliability of the hardware. In this paper we survey methods of determining the design correctness of systems as applied to computer programs.

Automated program proving techniques are still not practical for realistic programs. Manual proofs are lengthy, tedious, and error-prone. Software reliability provides a measure of confidence in the operational correctness of the software. Since the early 1970's several software reliability models have been proposed. We classify and discuss these models using the concepts of residual error size and the testing process used. We also discuss methods of estimating the correctness of the program and the adequacy of the set of test cases used.

These methods are directly applicable to assessing the design correctness of the total integrated hardware/software system which ultimately could include large complex distributed processing systems.

An excellent survey of reliability, with a comprehensive bibliography containing 114 references. Readers need to have a strong statistics background.

Rapps85

Rapps, Sandra, and Elaine J. Weyuker. "Selecting Software Test Data Using Data Flow Information." *IEEE Trans. Software Eng. SE-11*, 4 (April 1985), 367-375.

Abstract: This paper defines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria, which examine only the control flow of a program, are inadequate. Our procedure associates with each point in a program at which a variable is defined, those points at which the value is used. Several test data selection criteria, differing in the type and number of these associations, are defined and compared.

This paper explores the hierarchical relationships among several data flow testing techniques. The emphasis is on specifying criteria that should be satisfied by test data, not on generating the data.

The paper should be read by the instructor if data flow is to be treated in depth. The paper is likely to overwhelm students.

Redwine83

Redwine, Samuel T., Jr. "An Engineering Approach to Software Test Data Design." *IEEE Trans. Software Eng. SE-9*, 2 (March 1983), 191-200.

Abstract: A systematic approach to test data design is presented based on both practical translation of theory and organization of professional lore. The approach is organized around five domains and achieving coverage (exercise) of them by the test data. The domains are processing functions, input, output, interaction among functions, and the code itself. Checklists are used to generate data for processing functions. Separate checklists have been constructed for eight common business data processing functions such as editing, updating, sorting, and reporting. Checklists or specific concrete directions also exist for input, output, interaction, and code coverage. Two global heuristics concerning all test data are also used. A limited discussion on documenting test input data, expected results, and actual results is included.

Use, applicability, and possible expansions are covered briefly. Introduction of the method has similar difficulties to those experienced when intro-

ducing any disciplined technique into an area where discipline was previously lacking. The approach is felt to be easily modifiable and usable for types of systems other than the traditional business data processing ones for which it was originally developed.

This is one of the best articles directed toward a systematic means of testing data processing software. The value of this paper lies in its pragmatic approach to test data selection; there is little theory presented here.

As an example of applied testing in business applications, this paper is a winner. It could serve as a self-assessment test for students who have to develop an integrated method.

Richardson85

Richardson, Debra J., and Lori A. Clarke. "Partition Analysis: A Method Combining Testing and Verification." *IEEE Trans. Software Eng. SE-11*, 12 (Dec. 1985), 1477-1490.

Abstract: The partition analysis method compares a procedure's implementation to its specification, both to verify consistency between the two and to derive test data. Unlike most verification methods, partition analysis is applicable to a number of different types of specification languages, including both procedural and nonprocedural languages. It is thus applicable to high-level descriptions as well as to low-level designs. Partition analysis also improves upon existing testing criteria. These criteria usually consider only the implementation, but partition analysis selects test data that exercise both a procedure's intended behavior (as described in the specifications) and the structure of its implementation. To accomplish these goals, partition analysis divides or partitions *a procedure's domain into subdomains in which all elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This partition divides the procedure domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data and to verify consistency between the specification and the implementation. Moreover, the testing and verification processes are designed to enhance each other. Initial experimentation has shown that through the integration of testing and verification, as well as through the use of information derived from both the implementation and the specification, the partition analysis method is effective for evaluating program reliability. This paper describes the partition analysis method and reports the results obtained from an evaluation of its effectiveness.*

This paper contains an excellent presentation of a hybrid approach, in which simultaneous coverage of both code and specification is attempted. Prerequisite reading includes domain testing [White80] and symbolic execution and formal verification [Hantler76].

This paper is essential reading for both instructor and student.

Roussopoulos85

Roussopoulos, Nicolas, and Raymond T. Yeh. "SEES—A Software Testing Environment Support System." *IEEE Trans. Software Eng. SE-11*, 4 (April 1985), 355-366.

Abstract: SEES is a database system to support program testing. The program database is automatically created during the compilation of the program by a compiler built using the YACC compiler-compiler.

The database contains static information about the compiled program and is accessed via a relational database management system. SEES allows very flexible access to the data by selecting the level of detail and very powerful tools for writing reports tailored to the user needs.

This paper rediscovers the idea that the static analysis of programs is facilitated by building a relational database from information defined during compilation. Tool building is then done in the language supplied by the DBMS.

The paper could prove useful to the instructor as alternative technology that can be used for static analysis of programs.

Rowland81

Rowland, John H., and Philip J. Davis. "On the Use of Transcendentals for Program Testing." *J. ACM 28*, 1 (Jan. 1981), 181-190.

Abstract: The element z *is called a transcendental for the class* F *if functions in* F *can be uniquely identified by their values at* z. *Conditions for the existence of transcendentals are discussed for certain classes of polynomials, multinomials, and rational functions. Of particular interest are those transcendentals having an exact representation in computer arithmetic. Algorithms are presented for reconstruction of the coefficients of a polynomial from its value at a transcendental. The theory is illustrated by application to polynomials, quadratic forms, and quadrature formulas.*

This paper presents many techniques for demonstrating that a particular function has been implemented in a computer program. The paper requires a good background in functional analysis to grasp all the details. The paper is very well written, though it has limited application.

The paper can prove useful to the instructor, especially in gaining understanding of issues involved in selecting test data for particular program paths. It is not recommended for students.

Senn83

Senn, Edmond H., Kathy R. Ames, and Kathryn A. Smith. "Integrated Verification and Testing System (IVTS) for HAL/S Programs." *Proc. SOFTFAIR: A Conf. on Software Development Tools, Techniques, and Alternatives.* New York: IEEE, July 1983, 23-31.

Abstract: The IVTS is a large software system designed to support user-controlled verification analysis and testing activities for programs written in the HAL/S language. The system is composed of a user interface and user command language, analysis tools and an organized data base of host system files. The analysis tools are of four major types: (1) static analysis, (2) symbolic execution, (3) dynamic analysis (testing), and (4) documentation enhancement.

The IVTS requires a split HAL/S compiler, divided at the natural separation point between the parser/lexical analyzer phase and the target machine code generator phase. The IVTS uses the internal program form (HALMAT) between these two phases as primary input for the analysis tools. The dynamic analysis component requires some way to "execute" the object HAL/S program. The execution medium may be an interpretive simulation or an actual host or target machine.

IVTS represents state-of-the-art in testing integration. The paper is brief but presents an overview of the functionality of the system.

IVTS can serve as a useful example of integrated testing, but not enough material is available here for a full lecture.

Shneiderman79

Shneiderman, Ben. *Software Psychology: Human Factors in Computer and Information Systems.* Cambridge, Mass.: Winthrop Publishers, 1979.

This book describes empirical analysis performed on the programming process. It is useful for understanding how to conduct experiments into human factors issues associated with programming languages and interfaces.

Tai80

Tai, Kuo-Chung. "Program Testing Complexity and Test Criteria." *IEEE Trans. Software Eng. SE-6*, 6 (Nov. 1980), 531-538.

Abstract: This paper explores the testing complexity *of several classes of programs, where the testing complexity is measured in terms of the number of test data required for demonstrating program correctness by testing. It is shown that even for very restrictive classes of programs, none of the commonly used test criteria, namely, having every statement, branch, and path executed at least once, is nearly sufficient to guarantee absence of errors.*

Based on the study of testing complexity, this paper proposes two new test criteria, one for testing a path and the other for testing a program. These new criteria suggest how to select test data to obtain confidence in program correctness beyond the requirement of having each statement, branch, or path tested at least once.

This paper analyzes the complexity of achieving several structural coverage measures. The inadequacy of these measures is again shown, along with new criteria for demonstrating correctness for a limited class of programs.

The paper should be read by the instructor to gain an appreciation of when testing is equivalent to correctness. It is in-depth reading for a student interested in structural testing.

Weiser84

Weiser, Mark. "Program Slicing." *IEEE Trans. Software Eng. SE-10*, 4 (July 1984), 352-357.

Abstract: Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice," is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.

Some properties of slices are presented. In particular, finding statement-minimal *slices is in general unsolvable, but using data flow analysis is sufficient to find approximate slices. Potential applications include automatic slicing tools for debugging and parallel processing of slices.*

Though not directly related to program testing, this paper illustrates an application of data flow analysis with a strong intuitive appeal. Connections to symbolic execution and mutation testing can be readily made.

The paper could prove useful to the instructor and the student.

Weyuker80

Weyuker, Elaine J., and Thomas J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE Trans. Software Eng.*

SE-6, 3 (May 1980), 236-246. Reprinted in [Miller81a].

Abstract: The theory of test data selection proposed by Goodenough and Gerhart is examined. In order to extend and refine this theory, the concepts of a revealing test criterion and a revealing subdomain are proposed. These notions are then used to provide a basis for constructing program tests.

A subset of a program's input domain is revealing if the existence of one incorrectly processed input implies that all of the subset's elements are processed incorrectly. The intent of this notion is to partition the program's domain in such a way that all elements of an equivalence class are either processed correctly or incorrectly. A test set is then formed by choosing one element from each class. This process represents perfect program testing. For a practical testing strategy, the domain is partitioned into subdomains which are revealing for errors considered likely to occur.

Three programs which have previously appeared in the literature are discussed and tested using the notions developed in the paper.

This is the foundational paper for error-based testing. The criticism of [Goodenough75] is crisp, and the theoretical approach has established this as a classic paper.

This is essential reading for the instructor. The student who wishes to pursue error-based testing must read it also.

Weyuker82
Weyuker, Elaine J. "On Testing Non-testable Programs." *Computer J. 25*, 4 (Nov. 1982), 465-470.

*Abstract: A frequently invoked assumption in program testing is that there is an oracle (*i.e.*, the tester or an external mechanism can accurately decide whether or not the output produced by a program is correct). A program is non-testable if either an oracle does not exist or the tester must expend some extraordinary amount of time to determine whether or not the output is correct. The reasonableness of the oracle assumption is examined and the conclusions is reached that in many cases this is not a realistic assumption. The consequences of assuming the availability of an oracle are examined and alternatives investigated.*

Weyuker83
Weyuker, Elaine J. "Assessing Test Data Adequacy through Program Inference." *ACM Trans. Prog. Lang. and Syst. 5*, 4 (Oct. 1983), 641-655.

Abstract: Despite the almost universal reliance on testing as the means of locating software errors and its long history of use, few criteria have been proposed for deciding when software has been thoroughly tested. As a basis for the development of usable notions of test data adequacy, an abstract definition is proposed and examined, and approximations to this definition are considered.

Weyuker84
Weyuker, Elaine J. "The Complexity of Data Flow Criteria for Test Data Selection." *Information Processing Letters 19*, 2 (Aug. 1984), 103-109.

Weyuker86
Weyuker, Elaine J. "Axiomatizing Software Test Data Adequacy." *IEEE Trans. Software Eng. SE-12*, 12 (Dec. 1986), 1128-1138.

Abstract: A test data adequacy criterion is a set of rules used to determine whether or not sufficient testing has been performed. A general axiomatic theory of test data adequacy is developed, and five previously proposed adequacy criteria are examined to see which of the axioms are satisfied. It is shown that the axioms are consistent, but that only two of the criteria satisfy all of the axioms.

White80
White, Lee J., and Edward I. Cohen. "A Domain Strategy for Computer Program Testing." *IEEE Trans. Software Eng. SE-6*, 3 (May 1980), 247-257. Reprinted in [Miller81a].

Abstract: This paper presents a testing strategy designed to detect errors in the control flow of a computer program, and the conditions under which this strategy is reliable are given and characterized. The control flow statements in a computer program partition the input space into a set of mutually exclusive domains, *each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within ε of each boundary, under the appropriate assumptions, the strategy is shown to be reliable in detecting domain errors of magnitude greater than ε. Moreover, the number of test points required to test each domain grows only linearly with both the dimensionality of the input space and the number of predicates along the path being tested.*

This is the fundamental paper on domain testing, an error-based testing strategy. The paper focuses on testing errors in the control flow of programs whose predicates have linear interpretation in the input variables. Note that the restrictions specified in the

paper, especially linearity and the absence of arrays, limit the applicability of this strategy mostly to data processing programs. The strategy is examined closely in [Clarke82] and complemented by the approach in [Zeil83]. It is very well written and requires little background, though [Howden76] should probably be read first.

This paper is essential reading for the instructor and is very readable for students.

White85

White, Lee J. "Domain Testing and Several Outstanding Research Problems in Program Testing." *Infor/Canadian J. of Operational Res. & Info. Processing 23*, 1 (1985), 53-68.

Abstract: In the area of program testing, there are several significant problems which need to be addressed. It will be shown that a strategy called Domain Testing *can offer an approach when combined with the recent progress in software engineering. The problems include the determination of a scientifically sound basis for the selection of test data, the development of program specifications which can be used to both generate test data and also ascertain the correctness of program output, and the development of relationships between program testing and formal verification.*

This paper is a survey of research related to domain testing, with several suggestions of problems yet to be solved. The author attempts, with some success, a synthesis with related areas such as program verification, specification-based testing, and path selection criteria.

This well-written paper can serve as a general introduction to domain testing. It can be understood without prior knowledge of the subject. It can serve as a catalyst for encouraging detailed study in the area.

Woodward80

Woodward, Martin R., David Hedley, and Michael A. Hennell. "Experience with Path Analysis and Testing of Programs." *IEEE Trans. Software Eng. SE-6*, 3 (May 1980), 278-286. Reprinted in [Miller81a].

Abstract: There are a number of practical difficulties in performing a path testing strategy for computer programs. One problem is in deciding which paths, out of a possible infinity, to use as test cases. A hierarchy of structural test metrics is suggested to direct the choice and to monitor the coverage of test paths. Another problem is that many of the chosen paths may be infeasible in the sense that no test data can ever execute them. Experience with the use of "allegations" to circumvent this problem and prevent the static generation of many infeasible paths is reported.

This paper introduces the concept of LCSAJ, a linear code sequence and jump, which has since been used as a structural measure in several diverse experiments.

The paper should be read by the instructor interested in practical methods of structural testing. Students will find the paper difficult but rewarding.

WST88

Proc. 2nd Workshop on Software Testing, Analysis, and Verification. Washington, D. C.: IEEE Computer Society Press, July 1988.

Zeil83

Zeil, Steven J. "Testing for Perturbations of Program Statements." *IEEE Trans. Software Eng. SE-9*, 3 (May 1983), 335-346.

Abstract: Many testing methods require the selection of a set of paths on which tests are to be conducted. Errors in arithmetic expressions within program statements can be represented as perturbing functions added to the correct expression. It is then possible to derive the set of errors in a chosen functional class which cannot possibly be detected using a given test path. For example, test paths which pass through an assignment statement "X := f(Y)" *are incapable of revealing if the expression* "X - f(Y)" *has been added to later statements. In general, there are an infinite number of such undetectable error perturbations for any test path. However, when the chosen functional class of error expressions is a vector space, a finite characterization of all undetectable expressions can be found for one test path, or for combined testing along several paths. An analysis of the undetectable perturbations for sequential programs operating on integers and real numbers is presented which permits the detection of multinomial error terms. The reduction of the space of (potential) undetected errors is proposed as a criterion for test path selection.*

This paper describes a method for deducing sufficient path coverage to ensure the absence of prescribed errors in a program. It models the program computation and potential errors as a vector space. This enables the conditions for non-detection of an error to be calculated. The strategy assumes the existence of a reliable testing strategy for paths, which, of course, does not exist.

Exposure to [White80] should provide sufficient background for appreciating the context in which the techniques are to be used. The paper explores an interesting area and deserves to be read by the instructor. To understand the mathematics requires some background in linear algebra, especially if some of the references are to be read. The paper is advanced reading for the student.

UNLIMITED, UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-CM-9-1.2	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL *(If applicable)* SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS *(City, State and ZIP Code)* CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS *(City, State and ZIP Code)* ESD/AVS HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL *(If applicable)* ESD/ AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS *(City, State and ZIP Code)* CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE *(Include Security Classification)*
Unit Testing and Analysis

12. PERSONAL AUTHOR(S)
Larry J. Morell, College of William and Mary

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Yr., Mo., Day)*	15. PAGE COUNT
FINAL	FROM ______ TO ______	April 1989	34

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
software testing, module testing, unit testing, test method

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This module examines the techniques, assessment, and management of unit testing and analysis. Testing and analysis strategies are categorized according to whether their coverage goal is functional, structural, error-oriented, or a combination of these. Mastery of the material in this module allows the software engineer to define, conduct, and evaluate unit tests and analyses and to assess new techniques proposed in the literature.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER *(Include Area Code)* 412 268-7630 22c. OFFICE SYMBOL ESD/AVS (SEI JPO)

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE   UNLIMITED, UNCLASSIFIED

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials* package (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials* package (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

CM-1	[superseded by CM-19]
CM-2	Introduction to Software Design
CM-3	The Software Technical Review Process*
CM-4	Software Configuration Management*
CM-5	Information Protection
CM-6	Software Safety
CM-7	Assurance of Software Quality
CM-8	Formal Specification of Software*
CM-9	Unit Testing and Analysis
CM-10	Models of Software Evolution: Life Cycle and Process
CM-11	Software Specifications: A Framework
CM-12	Software Metrics
CM-13	Introduction to Software Verification and Validation
CM-14	Intellectual Property Protection for Software
CM-15	Software Development and Licensing Contracts
CM-16	Software Development Using VDM
CM-17	User Interface Development*
CM-18	[superseded by CM-23]
CM-19	Software Requirements
CM-20	Formal Verification of Programs
CM-21	Software Project Management
CM-22	Software Design Methods for Real-Time Systems*
CM-23	Technical Writing for Software Engineers
CM-24	Concepts of Concurrent Programming
CM-25	Language and System Support for Concurrent Programming*
CM-26	Understanding Program Dependencies

Educational Materials

EM-1	Software Maintenance Exercises for a Software Engineering Project Course
EM-2	APSE Interactive Monitor: An Artifact for Software Engineering Education
EM-3	Reading Computer Programs: Instructor's Guide and Exercises